Conception et Réalisation

D’une base de données

Merise • PowerAMC • SQL Server • T-SQL

Stéphane Grare





Conception et Réalisation

D'une base de données

Merise - PowerAMC - SQL Server - T-SQL

Stéphane Grare



Préface

Ce tutoriel se présente sous forme d'ouvrage avec pour objectif la réalisation d'une base de données sous Microsoft SQL Server en passant par la conception à l'aide de la méthode d'analyse Merise sous Power AMC. Il s'agit plus exactement d'un recueil et de notes de synthèses issues de différents supports.

La méthode Merise est une méthode d'analyse, de conception et de réalisation de systèmes d'informations informatisés. Power AMC est un logiciel de modélisation. Il permet de modéliser les traitements informatiques et leurs bases de données associées commercialisés par la société Sybase. Microsoft SQL Server est un système de gestion de base de données (abrégé en SGBD ou SGBDR pour "Système de Gestion de Base de Données Relationnelles") développé et commercialisé par la société Microsoft. Bien qu'il ait été initialement co-développé par Sybase et Microsoft, Ashton-Tate a également été associé à sa première version, sortie en 1989. Cette version est sortie sur les plateformes Unix et OS/2. Depuis, Microsoft a porté ce système de base de données sous Windows et il est désormais uniquement pris en charge par ce système.

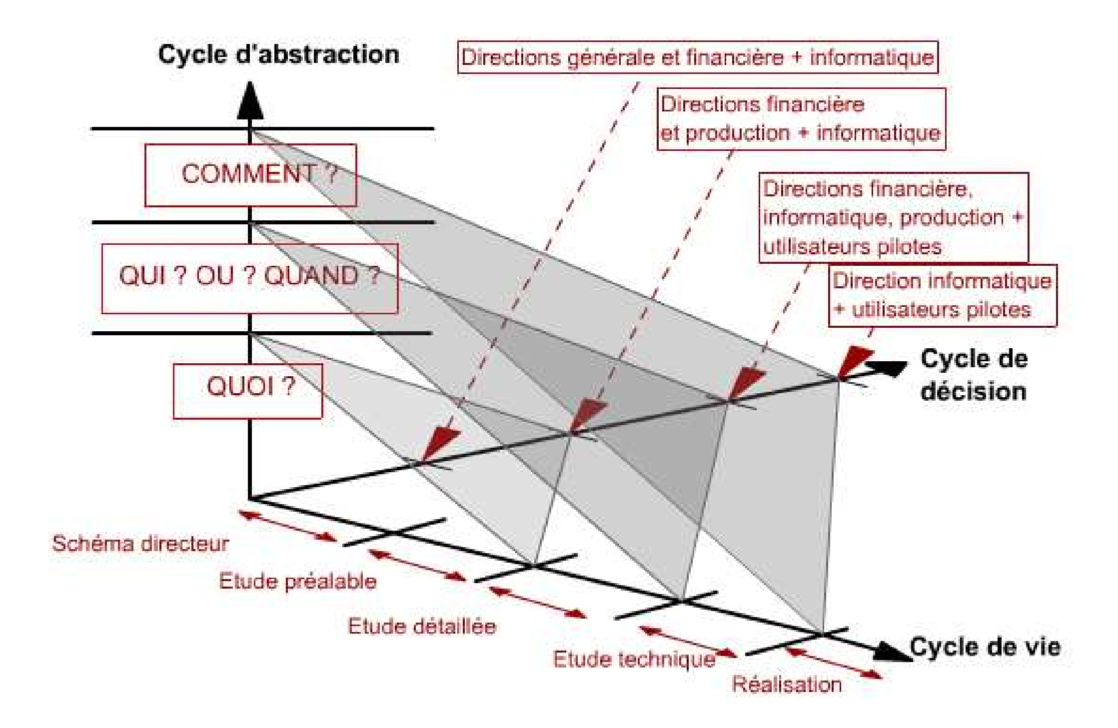
L'ouvrage se destine exclusivement aux étudiants de la formation professionnelle de l'Afpa, qui souhaitent apprendre et comprendre les grandes étapes nécessaires à la conception et à la réalisation d'une base de données. Il ne remplace en aucun cas les supports de formation nécessaire à l'apprentissage. Tout au long de l'ouvrage, nous utiliserons une base de données nommée « Papyrus ». Des exemples pourront porter sur des bases fictives afin d'apporter des notions supplémentaires.

Table des matières

Introduction à la méthode Merise

La méthode Merise est une méthode d'analyse, de conception et de réalisation de systèmes d'informations informatisés.

Merise part de l'idée selon laquelle la réalité dont elle doit rendre compte n'est pas linéaire, mais peut être définie comme la résultante d'une progression, menée de front, selon trois axes, qualifiées de "cycles".



La méthode Merise d'analyse et de conception propose une démarche articulée simultanément selon 3 axes pour hiérarchiser les préoccupations et les questions auxquelles répondre lors de la conduite d'un projet :

• **Cycle de vie** : Phases de conception, de réalisation, de maintenance puis nouveau cycle de projet.

• **Cycle de décision** : Des grands choix (GO-NO GO : Étude préalable), la définition du projet (étude détaillée) jusqu'aux petites décisions des détails de la réalisation et de la mise en œuvre du système d'information. Chaque étape est documentée et marquée par une prise de décision.

• **Cycle d'abstraction** : Niveaux conceptuels, logique / organisationnel et physique / opérationnel (du plus abstrait au plus concret). L'objectif du cycle d'abstraction est de prendre d'abord les grandes décisions métier, pour les principales activités (Conceptuel) sans rentrer dans le détail de questions d'ordre organisationnel ou technique.

Relativement à ces descriptions (encore appelées modèles) la méthode Merise préconise 3 niveaux d'abstraction :

- Le **niveau conceptuel** qui décrit la statique et la dynamique du système d'information en se préoccupant uniquement du point de vue du gestionnaire.

- Le **niveau organisationnel** décrit la nature des ressources qui sont utilisées pour supporter la description statique et dynamique du système d'information. Ces ressources peuvent être humaines et/ou matérielles et logicielles.

- Le **niveau opérationnel** dans lequel on choisit les techniques d'implantation du système d'information (données et traitements).

La conception du système d'information se fait par étapes, afin d'aboutir à un système d'information fonctionnelle reflétant une réalité physique. Il s'agit donc de valider une à une chacune des étapes en prenant en compte les résultats de la phase précédente. D'autre part, les données étant séparées des traitements, il faut vérifier la concordance entre données et traitement afin de vérifier que toutes les données nécessaires aux traitements sont présentes et qu'il n'y a pas de données superflues.

Cahier des charges

Nous allons présenter un exemple détaillé afin d'appréhender les différentes étapes de la conception à la réalisation d'une base de données auquel nous nous rapporterons tout au long de l'ouvrage.

Le souci majeur de M. Purchase, chef de la production informatique de la société Bidouille Express, est d'assurer la gestion et le suivi des produits *consommables* tels que :

- Papier listing en continu sous toutes ses formes,
- Papier pré imprimé (commandes, factures, bulletins de paie...)
- Rubans pour imprimantes
- Bandes magnétiques,
- Disquettes,
- ...

Pour chacun de ces produits, il existe plusieurs fournisseurs possibles ayant déjà livré la société ou avec lesquels M. Purchase est en contact. De plus, de nombreux représentants passent régulièrement vanter leurs produits et leurs conditions de vente : ceci permet à M. Purchase de conserver leurs coordonnées pour d'éventuelles futures commandes ou futurs appels d'offres. M. Purchase demande à chaque fournisseur ou représentant de lui proposer 3 tarifs différents en fonction de la quantité commandée et de mentionner leur délai de livraison.

Un degré de satisfaction est géré pour chaque fournisseur.

La commande est envoyée au fournisseur pour l'achat d'un ou plusieurs produits pour une quantité et un prix donnés. Cette quantité peut être livrée en plusieurs fois. Les seules informations mémorisées sont la date de dernière livraison ainsi que la quantité livrée totale.

Les règles de gestion

- Plusieurs fournisseurs ou représentants peuvent vendre le même produit à un prix fixé par le fournisseur, dépendant des quantités commandées (3 tranches de prix).
- Une commande est passée à un fournisseur ; elle se compose de plusieurs lignes, référençant chacune un produit.
- Le prix unitaire à la commande est fonction de la quantité commandée.

Conception de la base de données avec Power AMC

Power AMC est un logiciel de modélisation. Il permet de modéliser les traitements informatiques et leurs bases de données associées. Nous allons utilisez Power AMC pour la construction du Modèle Conceptuel de données à l'aide de la méthode Merise.

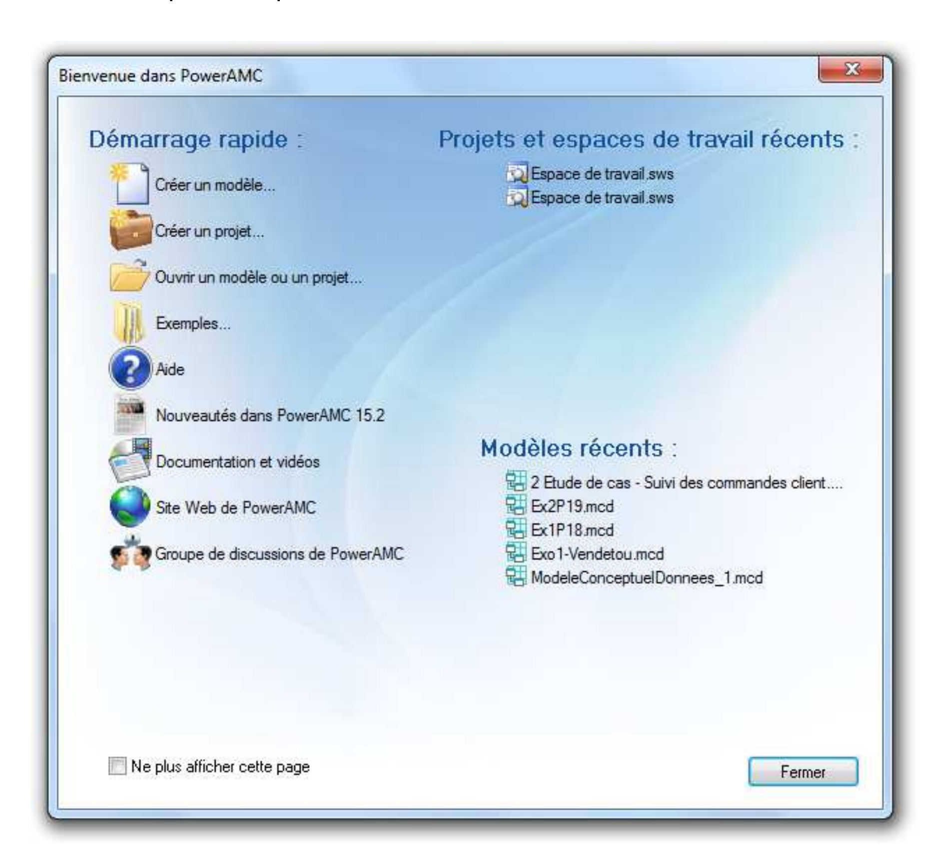
Au niveau conceptuel on veut décrire le modèle (le système) de l'entreprise ou de l'organisme :

• Le Modèle conceptuel des données (MCD), schéma représentant la structure du système d'information, du point de vue des données, c'est-à-dire les dépendances ou relations entre les différentes données du système d'information (par exemple : Le client, la commande, la ligne de commande...),

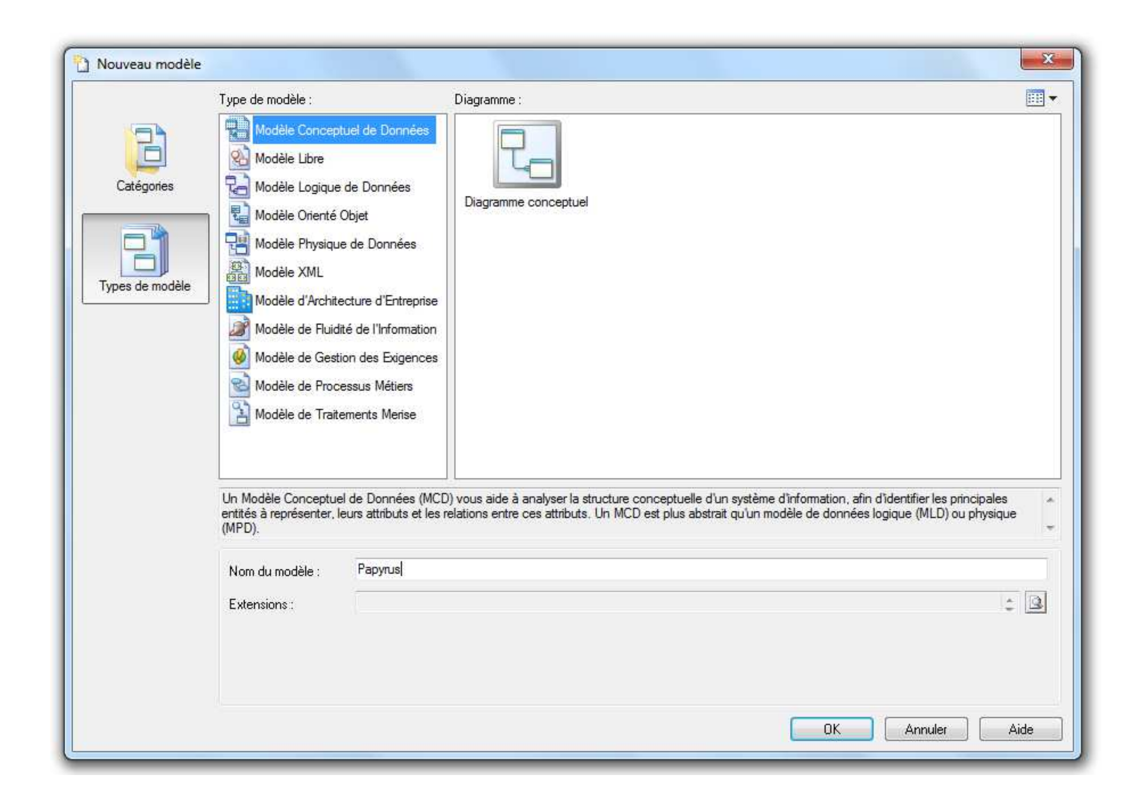
• Et le Modèle conceptuel des traitements (MCT), schéma représentant les traitements, en réponse aux événements à traiter (par exemple : La prise en compte de la commande d'un client).

Le MCD repose sur les notions d'entité et d'association et sur les notions de relations. Le MCT quant à lui est très peu utilisé est ne sera pas étudié au cours de ce tutoriel.

Pour créer un MCD avec Power AMC, créer un modèle directement à partir de l'écran de démarrage ou alors en passant par le menu : Fichier / Nouveau modèle…

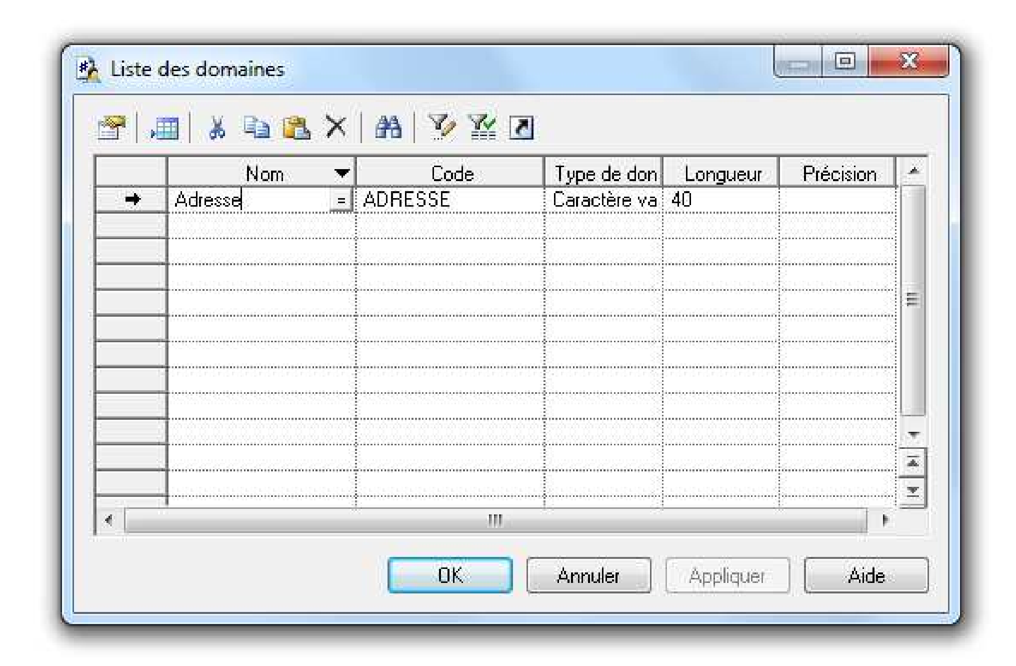


Dans « **Type de modèle**, sélectionnez « **Modèle Conceptuel de Données** » puis « **Diagramme Conceptuel** ». Nous nommerons notre exemple « Papyrus ».



Créer des domaines

Vous avez la possibilité de créer un type de données pour des champs déterminés sous Power AMC par le menu « **Modèle** » / « **Domaine** ». Par exemple, on peut définir un champ Adresse qui correspondra toujours au type de données défini (dans l'exemple, notre champ Adresse sera de type caractère variable de longueur 40).

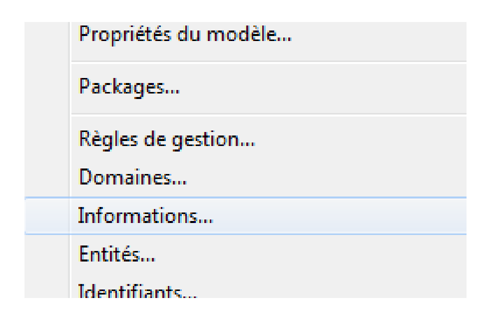


Le dictionnaire des données

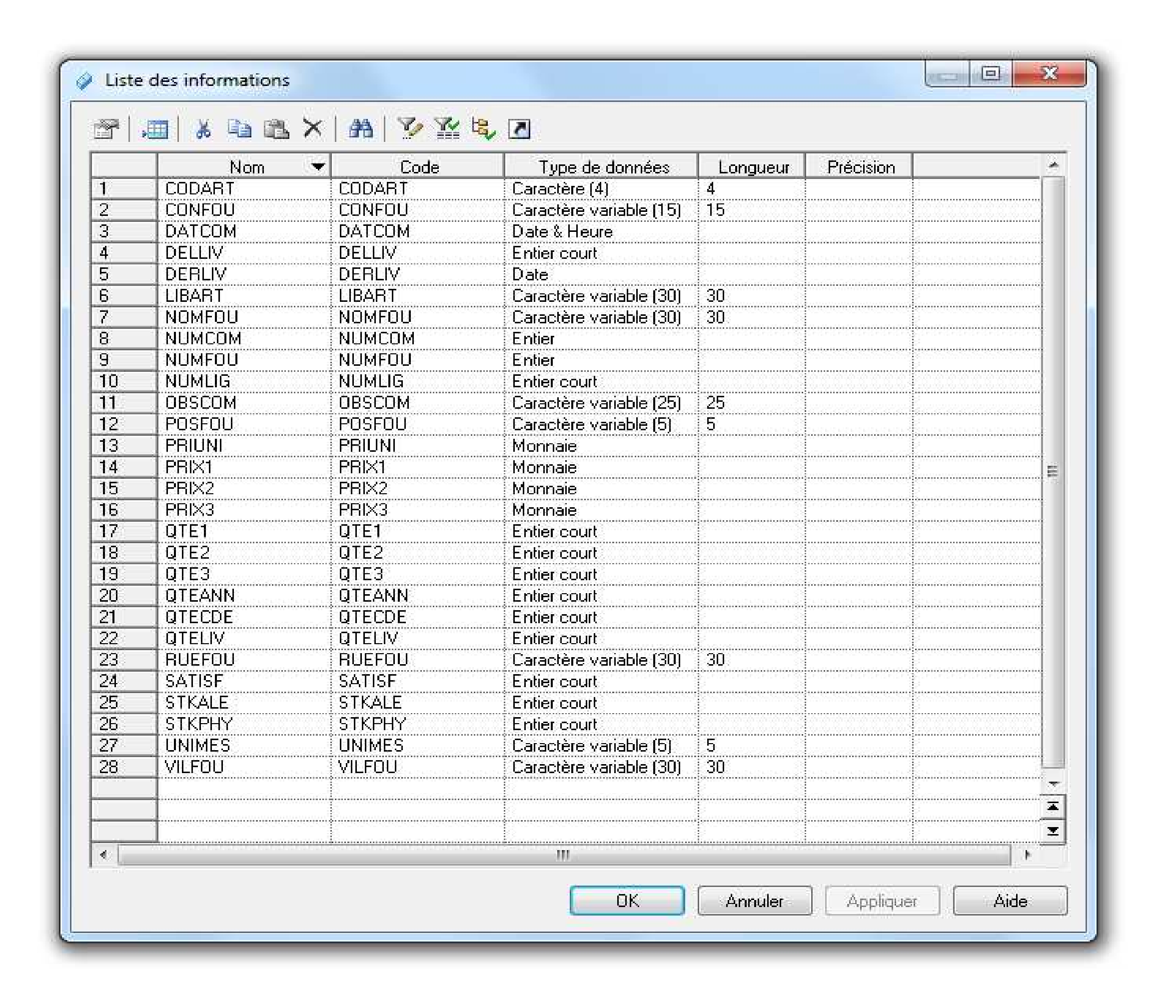
Les champs utilisés dans les différentes entités sont listés dans le tableau ci-dessous :

CODART	Code produit	char(4)
CONFOU	Contact chez le fournisseur	varchar(15)
DATCOM	Date de commande	smalldatetime
DELLIV	Délai de livraison	smallint
DERLIV	Date dernière livraison	date
LIBART	Libellé Produit	varchar(30)
NUMCOM	Numéro de commande	int
NUMFOU	N° de compte fournisseur	int
NUMLIG	N° de ligne commande	tinyint
NOMFOU	Nom fournisseur	varchar(30)
OBSCOM	Observations	varchar(25)
POSFOU	Code postal fournisseur	char(5)
PRIUNI	Prix unitaire de vente	smallmoney
PRIX1	Prix unitaire 1	smallmoney
PRIX2	Prix unitaire 2	smallmoney
PRIX3	Prix unitaire 3	smallmoney
QTE1	Borne quantité livraison 1	smallint
QTE2	Borne quantité livraison 2	smallint
QTE3	Borne quantité livraison 3	smallint
QTEANN	Quantité annuelle	smallint
QTECDE	Quantité commandée	smallint
QTELIV	Quantité livrée	smallint
RUEFOU	Adresse fournisseur	varchar(30)
SATISF	Indice satisfaction	tinyint
STKALE	Stock d'alerte	smallint
STKPHY	Stock physique	smallint
UNIMES	Unité de mesure	char(5)
VILFOU	Ville fournisseur	varchar(30)

Pour accéder aux dictionnaires des données avec Power AMC, utilisez le menu. Cliquez sur « **Modèle** » puis sur « **Informations** ».

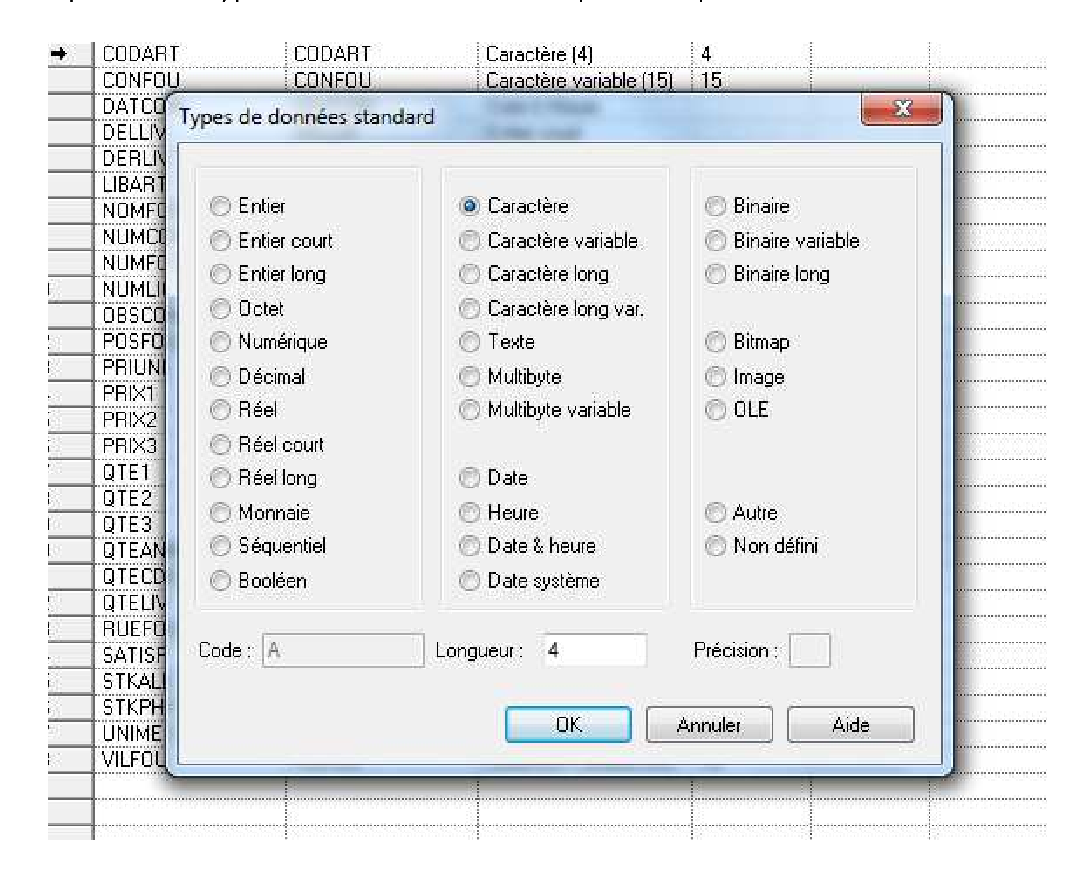


Vous pouvez alors remplir la liste comme suit :



	Nom	Code	Type de données	Longueur	Précision
1	CODART	CODART	Caractère (4)	4	
2	CONFOU	CONFOU	Caractère variable (15)	15	
3	DATCOM	DATCOM	Date & Heure		
4	DELLIV	DELLIV	Entier court		
5	DERLIV	DERLIV	Date		
6	LIBART	LIBART	Caractère variable (30)	30	
7	NOMFOU	NOMFOU	Caractère variable (30)	30	
8	NUMCOM	NUMCOM	Entier		
9	NUMFOU	NUMFOU	Entier		
10	NUMLIG	NUMLIG	Entier court		
11	OBSCOM	OBSCOM	Caractère variable (25)	25	
12	POSFOU	POSFOU	Caractère variable (5)	5	
13	PRIUNI	PRIUNI	Monnaie		
14	PRIX1	PRIX1	Monnaie		
15	PRIX2	PRIX2	Monnaie		
16	PRIX3	PRIX3	Monnaie		
17	QTE1	QTE1	Entier court		
18	QTE2	QTE2	Entier court		
19	QTE3	QTE3	Entier court		
20	QTEANN	QTEANN	Entier court		
21	QTECDE	QTECDE	Entier court		
22	QTELIV	QTELIV	Entier court		
23	RUEFOU	RUEFOU	Caractère variable (30)	30	
24	SATISF	SATISF	Entier court		
25	STKALE	STKALE	Entier court		
26	STKPHY	STKPHY	Entier court		
27	UNIMES	UNIMES	Caractère variable (5)	5	
28	VILFOU	VILFOU	Caractère variable (30)	30	

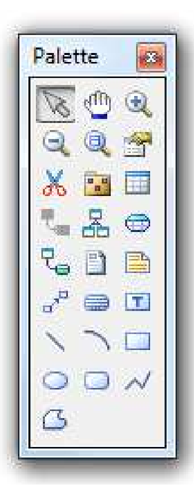
On devra préciser le type des données attendues pour chaque attribut.



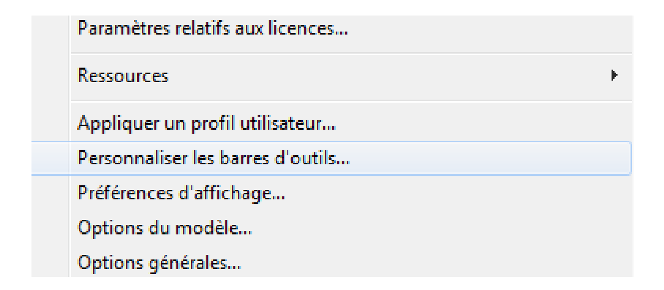
À noter que dans le cas où vous ne le remplissez pas, celui-ci se remplira automatiquement au fur à mesure que nous compléterons notre modèle.

Utilisation de la palette

On utilisera la palette pour positionner les différents éléments qui composeront votre modèle conceptuel des données.



PS : Si celle-ci n'apparaît pas à l'écran, vous avez la possibilité de l'afficher en utilisant le menu « **Outils** » puis « **Personnaliser les barres d'outils…** ».

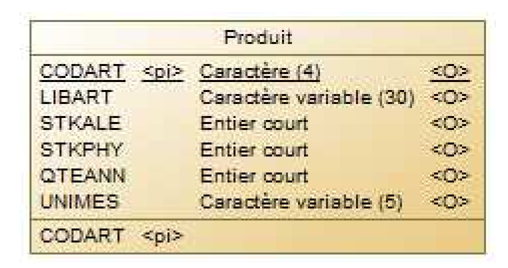


Vous devez alors cliquer sur la case « **Palette** » pour pouvoir l'afficher.

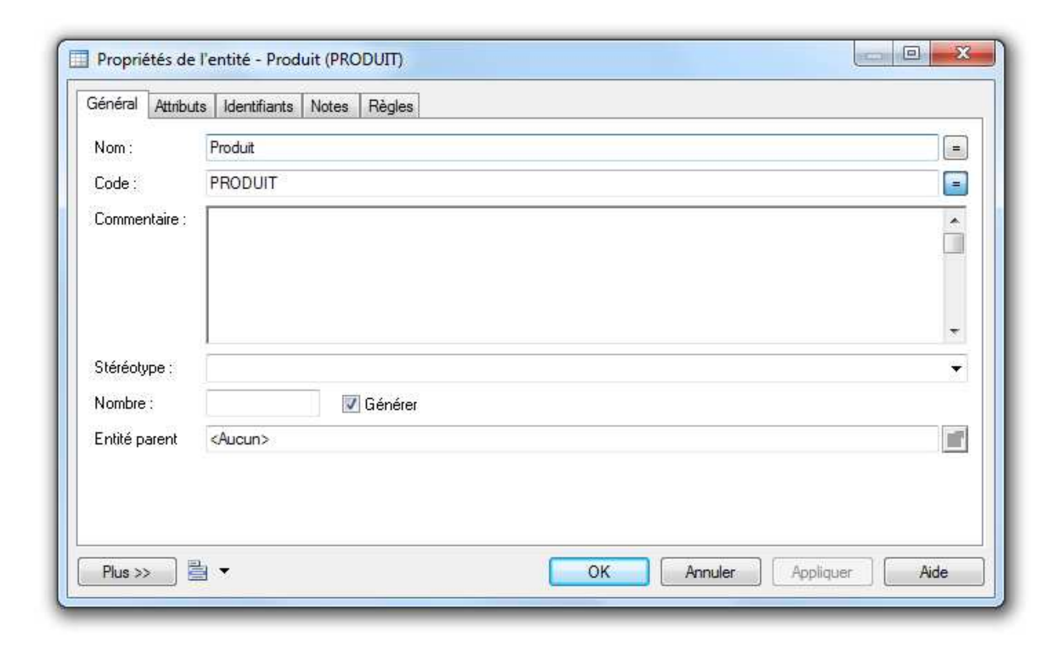


L'entité et les attributs

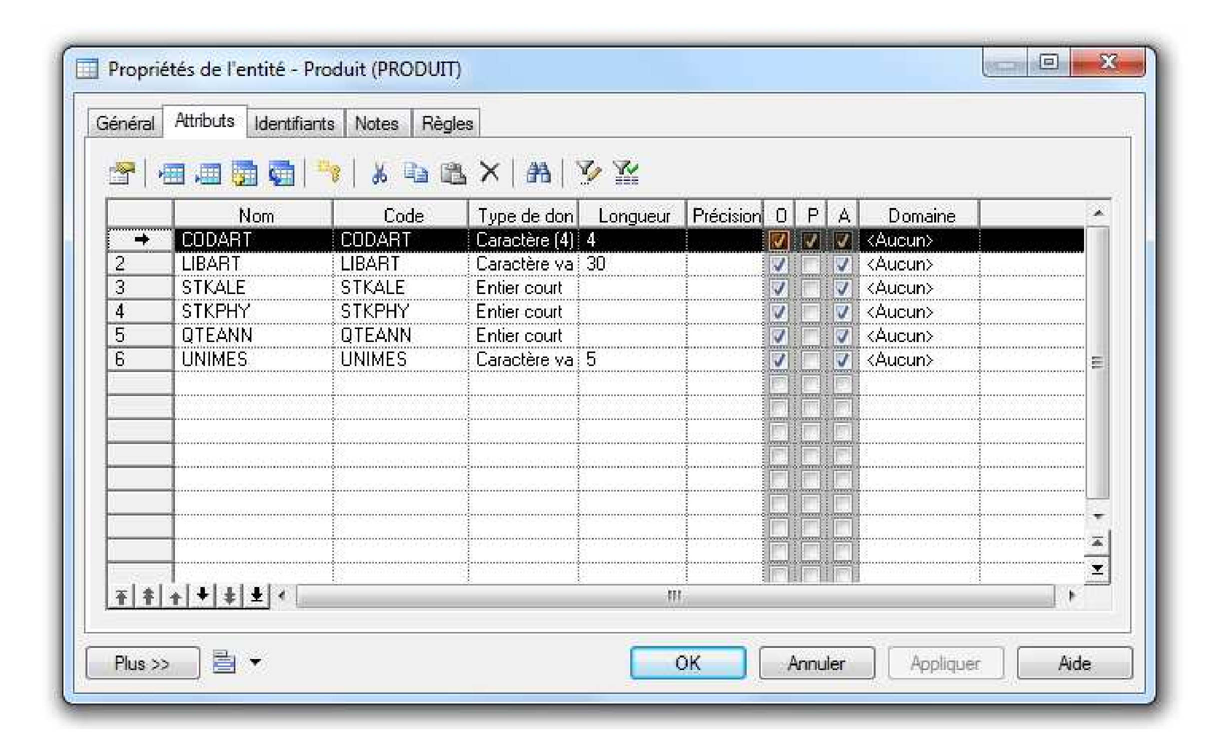
L'entité est définie comme un objet de gestion considéré d'intérêt pour représenter l'activité à modéliser (exemple : entité Produit) et chaque entité est porteuse d'une ou plusieurs propriétés simples (appelé attributs), dites atomiques. Exemples : CODART (qui représentera le code du produit), LIBART (libellé du produit)...



Pour compléter votre entité, cliquez droit / propriété.

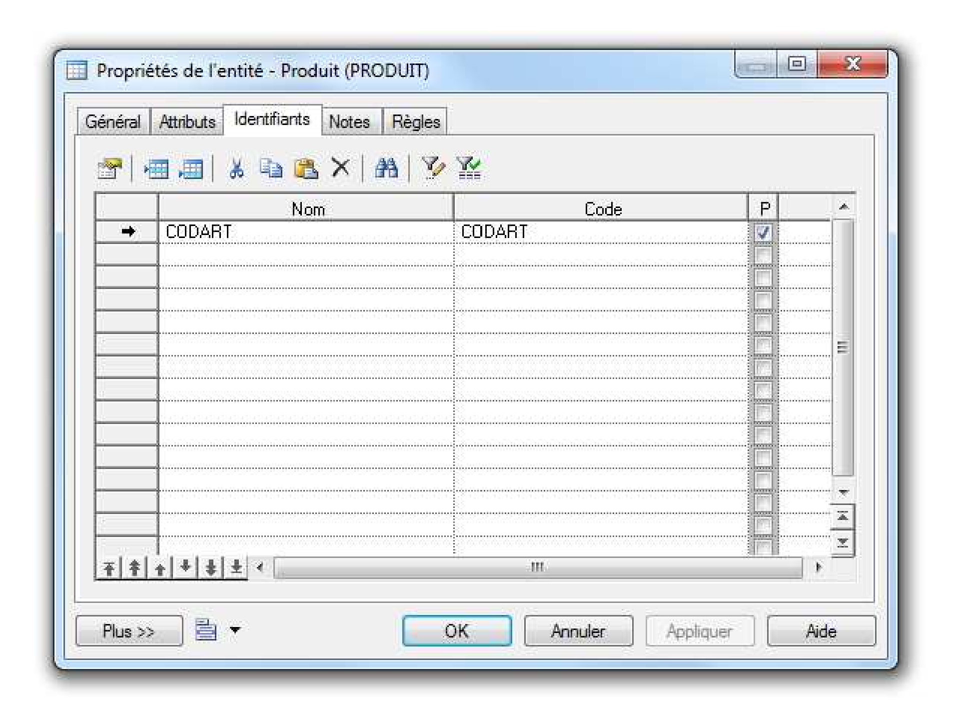


Sur l'onglet « **Général** », on indique le nom de notre entité. Pour compléter nos différents attributs, cliquez sur l'onglet « **Attributs** ».

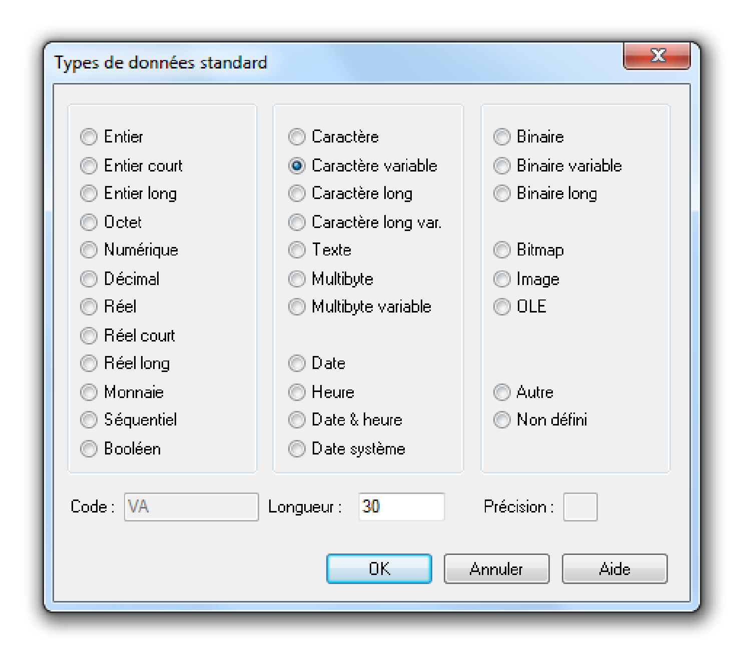


Si vous avez créé un domaine, alors c'est ici que vous devez l'indiquer. Le type de données se sélectionne alors automatiquement.

On cochera les cases « **O** » s'il s'agit d'une donnée obligatoire ou facultative et « **P** » pour indiquer la clé primaire. Une entité doit en effet posséder une propriété unique et discriminante, qui est désignée comme identifiant (exemple : CODART). On reportera cette information sur l'onglet « **Identifiants** ».



On devra préciser le type des données attendues pour chaque attribut.

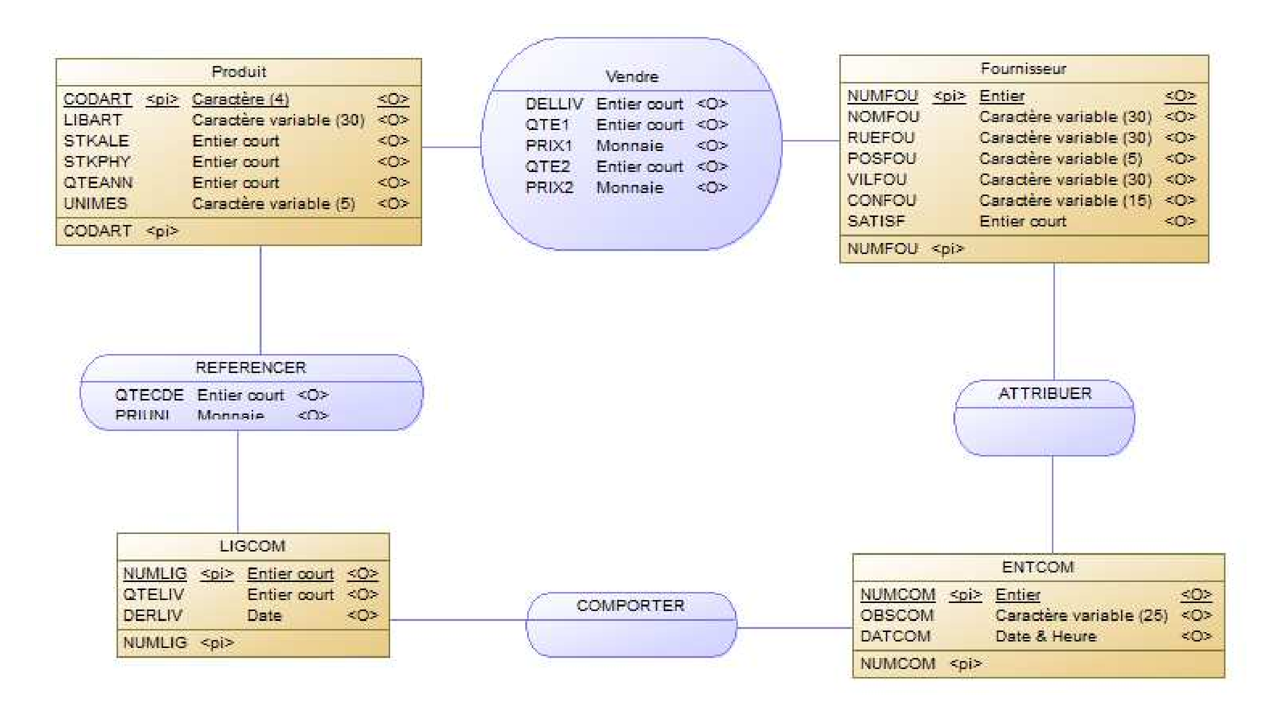


Par construction, le MCD impose que tous les attributs d'une entité aient vocation à être renseignées (il n'y a pas d'attribut « facultatif »). Les informations calculées (ex: montant taxes comprises d'une facture), déductibles (ex: densité démographique = population / superficie) ne doivent pas y figurer.

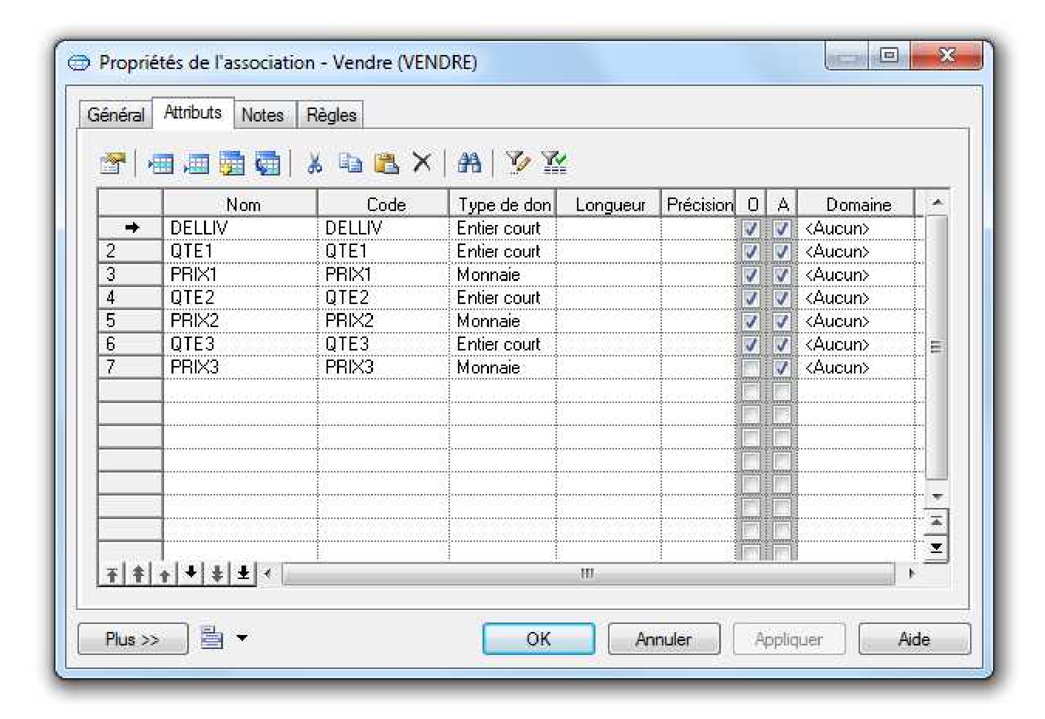
L'association (ou relation)

Ce sont des liaisons logiques entre les entités. Elles peuvent être de nature factuelle, ou de nature dynamique. Par exemple, une personne peut acheter un objet (action d'acheter), mais si l'on considère qu'une personne est propriétaire d'un objet, alors l'association entre l'objet et cette personne est purement factuelle.

Les associations se représentent dans une ellipse (ou un rectangle aux extrémités rondes), reliée par des traits aux entités qu'elles lient logiquement.

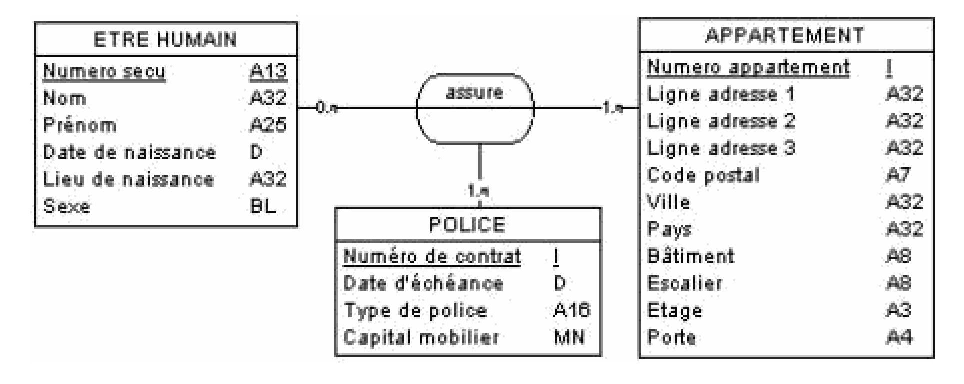


Une association peut elle aussi contenir des attributs.

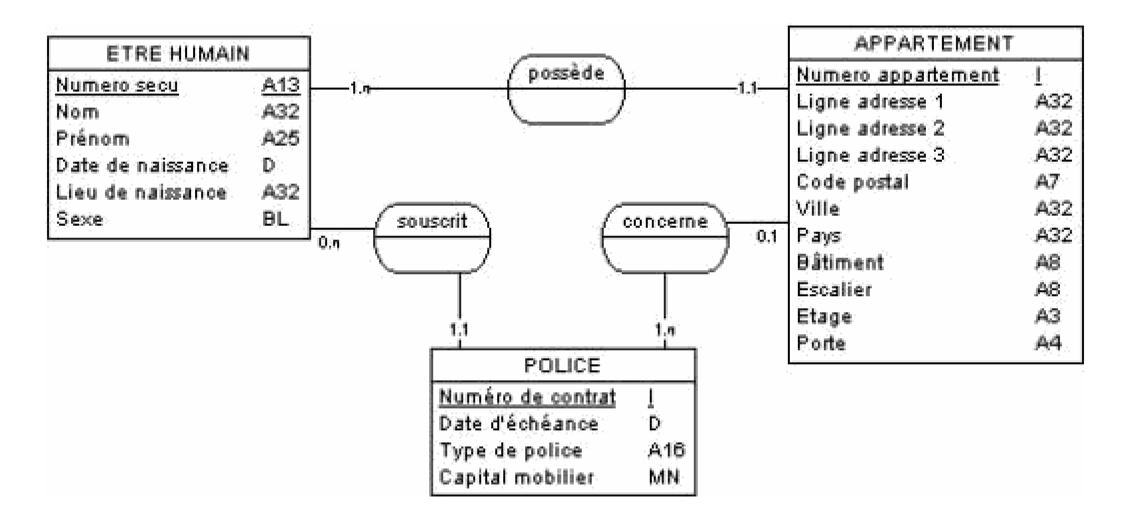


La plupart des associations sont de nature binaire, c'est à dire composées de deux entités mises en relation par une ou plusieurs associations. C'est le cas par exemple de l'association "est propriétaire" mettant en relation "être humain" et "appartement".

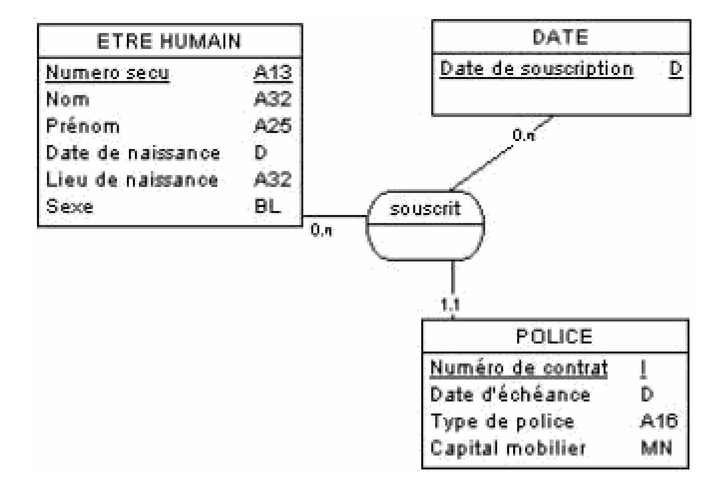
Cependant il arrive qu'une association concerne plus de deux entités (on dit alors qu'il s'agit d'association "n-aires").



Mais dans ce cas il y a de grandes difficultés pour exprimer les cardinalités. On aura tout intérêt à essayer de transformer le schéma de manière à n'obtenir que des associations binaires.

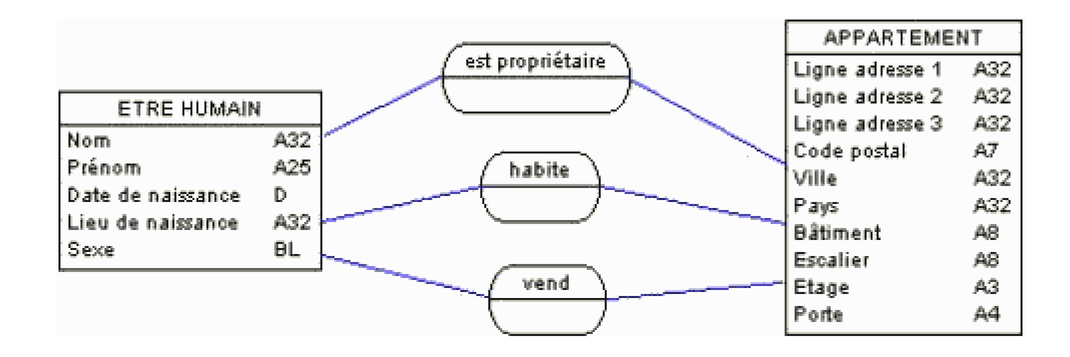


Il arrive dans certains cas que l'attribut "date" soit d'une importance capitale, notamment dans les applications SGBDR portant sur la signature de contrats à échéance ou dans la durée (assurance par exemple). Il n'est pas rare alors que le seul attribut "date" constitue à lui seul une entité.

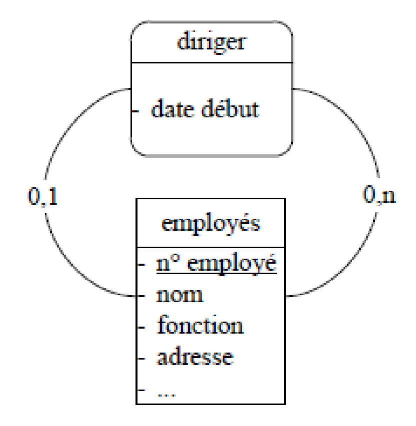


On appelle alors cela une entité temporelle. Une entité temporelle possède souvent un seul attribut, mais dans le cas où elle possède plusieurs attributs (année, mois, jour, heure, minute, seconde...), l'ensemble de ces attributs constitue alors la clef de l'entité.

Mais dans ce cas on peut aussi retirer cette entité et introduire la date en tant qu'attribut de l'association "souscrit". Deux mêmes entités peuvent être plusieurs fois en association.



Il est permis à une association d'être branchée plusieurs fois à la même entité, par exemple l'association binaire réflective suivante :

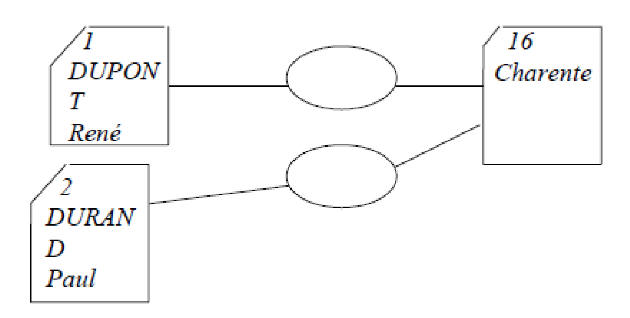


En résumé

. Une classe de relation **récursive** (ou *réflexive*) relie la même classe d'entité
. Une classe de relation **binaire** relie deux classes d'entité
. Une classe de relation **ternaire** relie trois classes d'entité
. Une classe de relation **n-aire** relie n classes d'entité

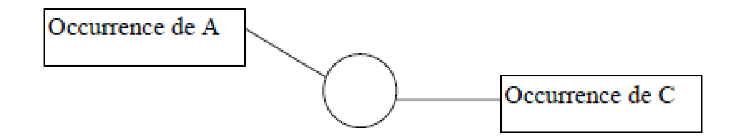
La dimension d'une association indique le nombre d'entités participant à l'association. Les dimensions les plus courantes sont 2 (association binaire) et 3 (association ternaire)

Une occurrence d'association est un lien particulier qui relie deux occurrences d'entités. Le schéma ci-dessous présente deux exemples d'occurrences de l'association « Habite ».

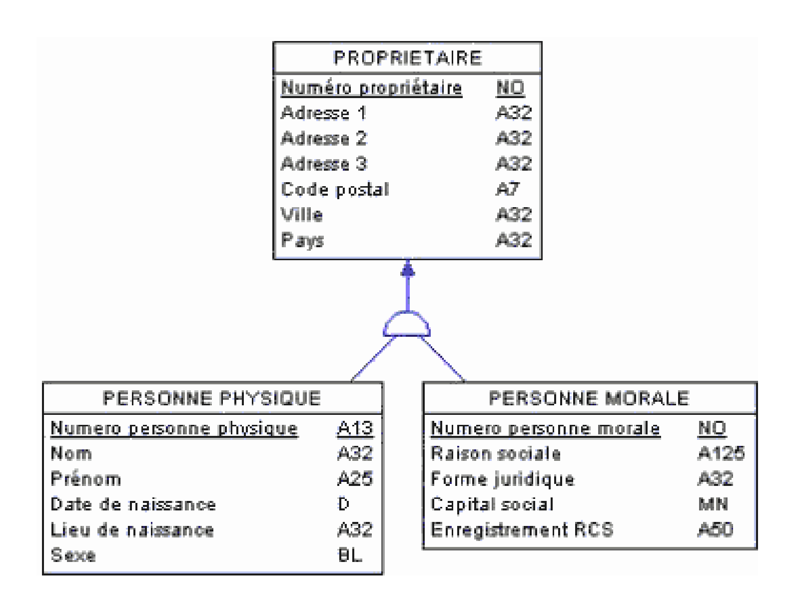


Remarque : Certains auteurs définissent l'identifiant d'une association comme étant la concaténation des identifiants des entités qui participent à l'association.

Toute occurrence d'une association de dimension n doit être reliée à n occurrences d'entités. Par exemple, pour une association ternaire dans laquelle participent trois entités « A », « B » et « C », toute occurrence doit être reliées à 3 occurrences des entités respectives A, B et C. On ne peut donc pas avoir une occurrence à 2 pattes de la forme ci-dessous.



Dans le schéma ci-dessous, les entités "Personne physique" (des êtres humains) et "Personne morale" (des sociétés, associations, collectivités, organisations…) sont généralisées dans l'entité "Propriétaire". On dit aussi que l'entité "Propriétaire" est une entité parente et que les entités "Personne morale" et "Personne physique" sont des entités enfants, car il y a une notion d'héritage...



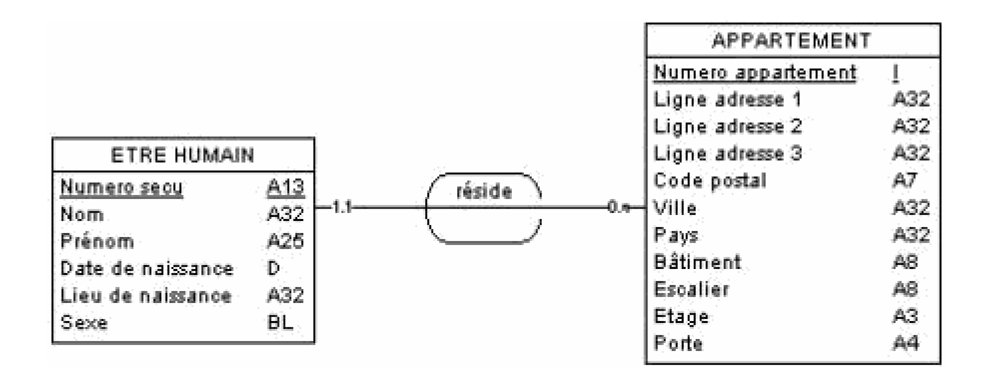
Par exemple une entité "Être humain" est une généralisation pour toute entité faisant appel à une personne, comme les entités "Étudiant", "Client", "Artiste", "Souscripteur", "Patient", "Assujetti"... On les appelle aussi "entités-génériques". Certains ateliers de modélisation représentant les données sous la forme d'entités « encapsulées ».

Les cardinalités

Les cardinalités permettent de caractériser le lien qui existe entre une entité et la relation à laquelle elle est reliée. La cardinalité d'une relation est composée d'un couple comportant une borne maximale et une borne minimale, intervalle dans lequel la cardinalité d'une entité peut prendre sa valeur :

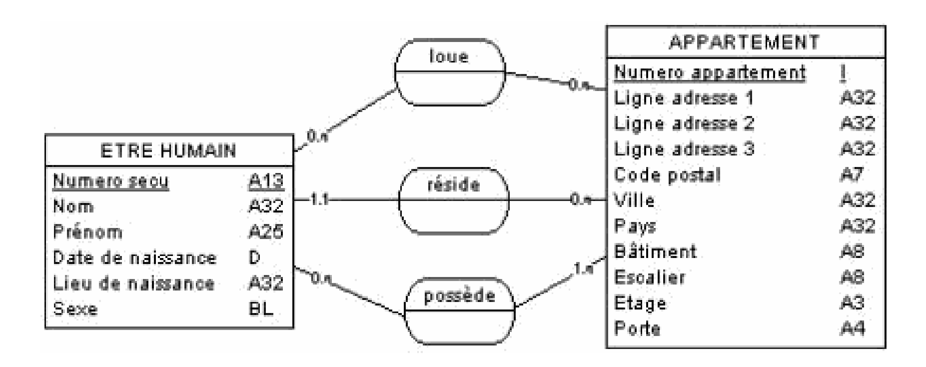
. La borne minimale (généralement 0 ou 1) décrit le nombre minimum de fois qu'une entité peut participer à une relation
. La borne maximale (généralement 1 ou n) décrit le nombre maximum de fois qu'une entité peut participer à une relation

On note les cardinalités de chaque côté de l'association, sur les traits faisant la liaison entre l'association et l'entité.



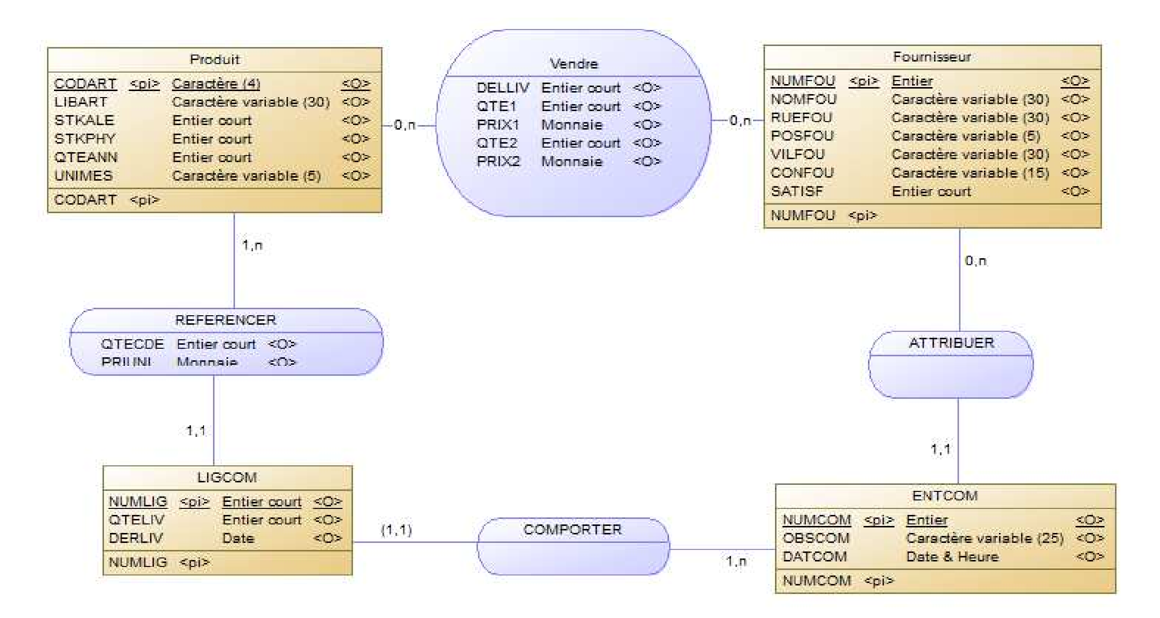
Dans l'exemple précédent, tout employé est dirigé par un autre employé (sauf le directeur d'où le 0,n) et un employé peut diriger plusieurs autres employés, ce qui explique les cardinalités sur le schéma.

Des relations différentes entre mêmes entités peuvent posséder des cardinalités différentes. C'est même souvent le cas.



La relation « loue » est de type n:m.
La relation « réside » est de type 1:n.
La relation « possède » est de type n:m.

Pour revenir à notre cas pratique, on en déduit les cardinalités suivantes :



Notion d'identifiant relatif

Les identifiants relatifs font partie des extensions Merise/2. Certaines entités ont une existence totalement dépendante d'autres entités. Dans ce cas nous avons recours à un identifiant relatif. Dans le schéma ci-dessus, nous avons un identifiant relatif entre l'entité LIGCOM et l'association COMPORTER qui s'écrit toujours avec une cardinalité (1,1). Une ligne de commande ne peut exister s'il elle ne comporte pas un numéro de commande.

Règles de normalisation

. **Normalisation des entités** : Toutes les entités qui sont remplaçables par une association doivent être remplacées.

. **Normalisation des noms** : Chaque entité doit posséder un identifiant qui caractérise ses individus de manière unique. Le nom d'une entité, d'une association ou d'un attribut doit être unique.

. **Normalisations des identifiants** : L'identifiant peut être composé de plusieurs attributs, mais les autres attributs de l'entité doivent être dépendant de l'identifiant en entier (et non

pas une partie de cet identifiant). Ces deux premières formes normales peuvent être oubliées si on suit le conseil de n'utiliser que des identifiants non composés de type numéro.

. **Normalisation des attributs des associations :** Les attributs d'une entité doivent dépendre directement de son identifiant. Par exemple, la date de fête d'un client ne dépend pas de son identifiant numéro de client, mais plutôt de son prénom. Elle ne doit pas figurer dans l'entité clients, il faut donc faire une entité « calendrier » à part, en association avec clients.

En effet, d'une part, les attributs en plusieurs exemplaires posent des problèmes d'évolutivité du modèle (comment faire si un employé à deux adresse secondaires ?) et d'autre part, les attributs calculables induisent un risque d'incohérence entre les valeurs des attributs de base et celles des attributs calculés.

. **Normalisation des associations :** Les attributs des associations doivent dépendre des identifiants de toutes les entités en association. Par exemple, la quantité commandée dépend à la fois du numéro de client et du numéro d'article, par contre la date de commande non.

. **Normalisation des cardinalités :** Il faut éliminer les associations fantômes, redondantes ou en plusieurs exemplaires.

Le modèle logique des données (MLD)

La transcription d'un MCD en modèle relationnel s'effectue selon quelques règles simples qui consistent d'abord à transformer toute entité en table, avec l'identifiant comme clé primaire, puis à observer les valeurs prises par les cardinalités maximums de chaque association pour représenter celle-ci soit (ex : card. max 1-n ou 0-n) par l'ajout d'une clé étrangère dans une table existante, soit (ex : card. max n-n) par la création d'une nouvelle table dont la clé primaire est obtenue par concaténation de clés étrangères correspondant aux entités liées.

Règle n°1 : Toute entité doit être représentée par une table. Toute entité devient une table dans laquelle les attributs deviennent des colonnes. L'identifiant de l'entité constitue alors la clé primaire de la table.

Règle n°2 : Dans le cas d'entités reliées par des associations de type 1:1, les tables doivent avoir la même clef.

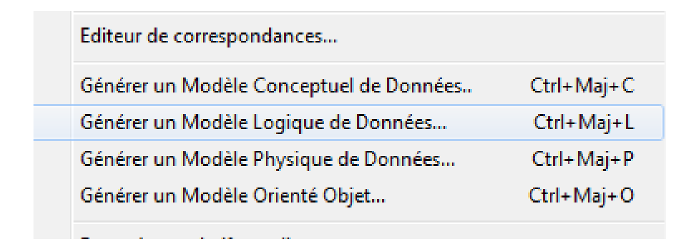
Règle n°3 : Dans le cas d'entités reliées par des associations de type 1:n, chaque table possède sa propre clef, mais la clef de l'entité côté 0,n (ou 1,n) migre vers la table côté 0,1 (ou 1,1) et devient une clef étrangère (index secondaire).

Règle n°4 : Dans le cas d'entités reliées par des associations de type n:m, une table intermédiaire dite table de jointure, doit être créée, et doit posséder comme clef primaire une conjonction des clefs primaires des deux tables pour lesquelles elle sert de jointure.

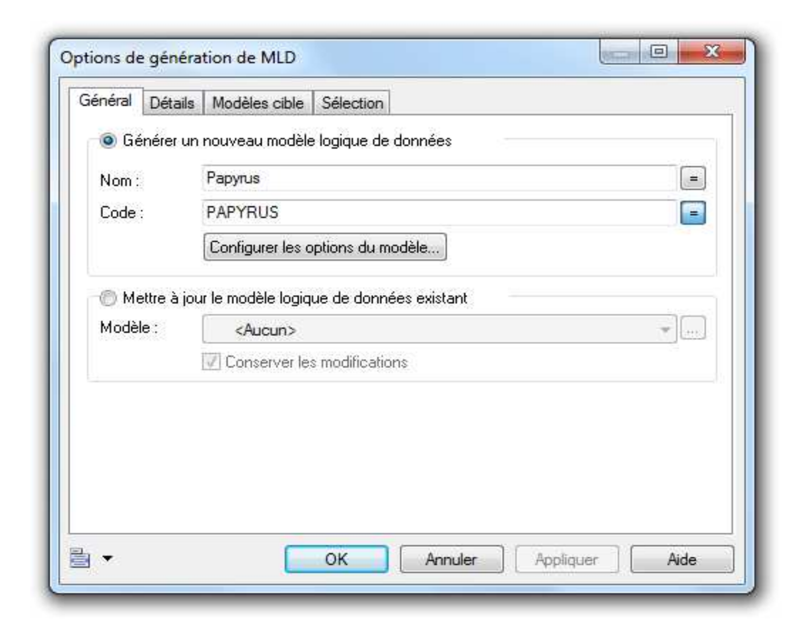
Règle n°5 : Cas des associations pourvues d'au moins un attribut :

. Si le type de relation est n:m, alors les attributs de l'association deviennent des attributs de la table de jointure.
. Si le type de relation est 1:n, il convient de faire glisser les attributs vers l'entité pourvue des cardinalités 1:1.
. Si le type de relation est 1:1, il convient de faire glisser les attributs vers l'une ou l'autre des entités.

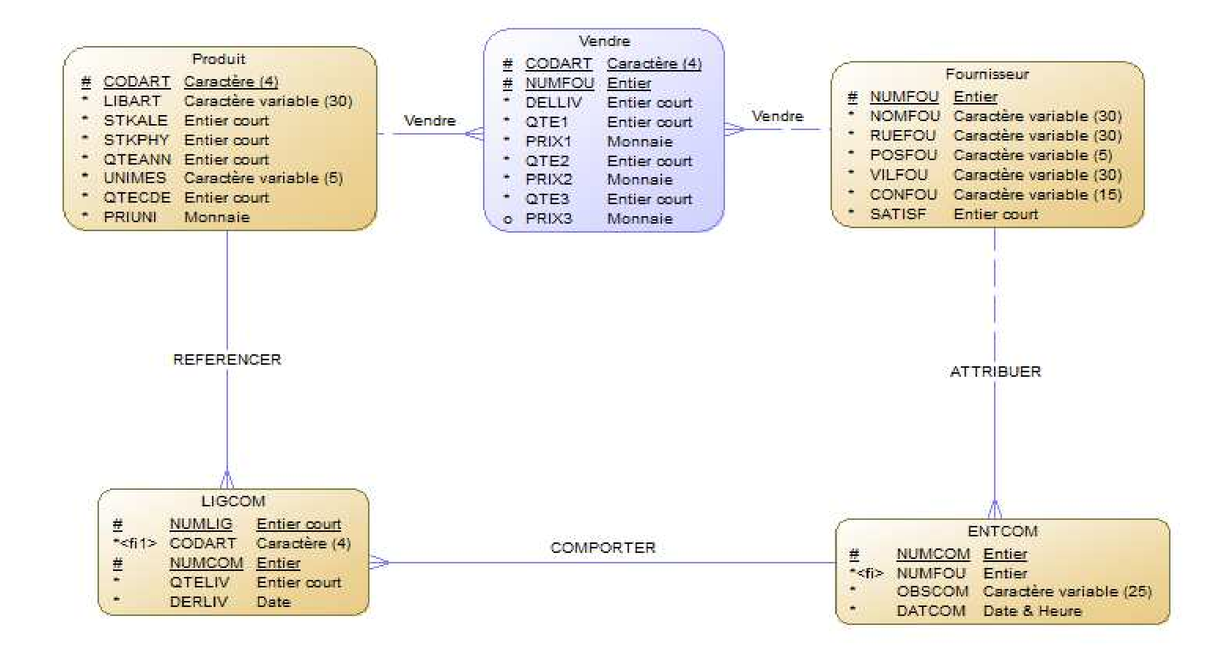
Avec Power AMC, le modèle logique de données se génère automatiquement si vous avez préalablement complété comme il se doit votre Modèle Conceptuel de données. Cliquez sur « **Outils** » de la barre de menu puis « **Générer un Modèle Logique de Données...** ».



Des options de configurations sont alors disponibles :



Cliquez sur « Ok » pour générer le MLD :



La base de données relationnelle PAPYRUS est constituée des relations suivantes :

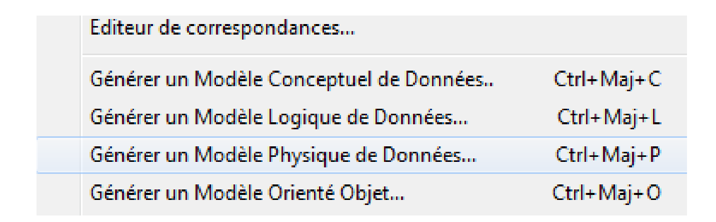
PRODUIT (CODART, LIBART, STKLE, STKPHY, QTEANN, UNIMES)
ENTCOM (NUMCOM, OBSCOM, DATCOM, NUMFOU)
LIGCOM (NUMCOM, NUMLIG, CODART, QTECDE, PRIUNI, QTELIV, DERLIV)
FOURNIS (NUMFOU, NOMFOU, RUEFOU, POSFOU, VILFOU, CONFOU, SATISF)
VENDRE (CODART, NUMFOU, DELLIV, QTE1, PRIX1, QTE2, PRIX2, QTE3, PRIX3)

Modèle physique de données (MPD)

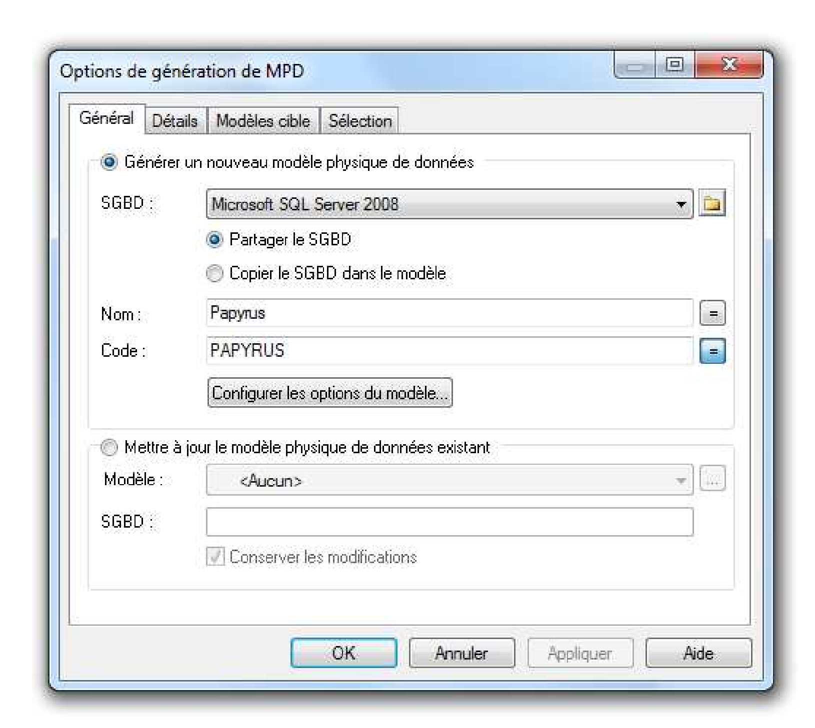
Le MPD est une implémentation particulière du MLD pour un matériel, un environnement et un logiciel donné. Notamment, le MPD s'intéresse au stockage des données à travers le type et la taille (en octets ou en bits) des attributs du MCD. Cela permet de prévoir la place nécessaire à chaque table dans le cas d'un SGBDR.

Le MPD tient compte des limites matérielles et logicielles afin d'optimiser l'espace consommé et d'optimiser le temps de calcul (qui représentent deux optimisations contradictoires). Dans le cas d'un SGBDR, le MPD définit les index et peut être amené à accepter certaines redondances d'information afin d'accélérer les requêtes.

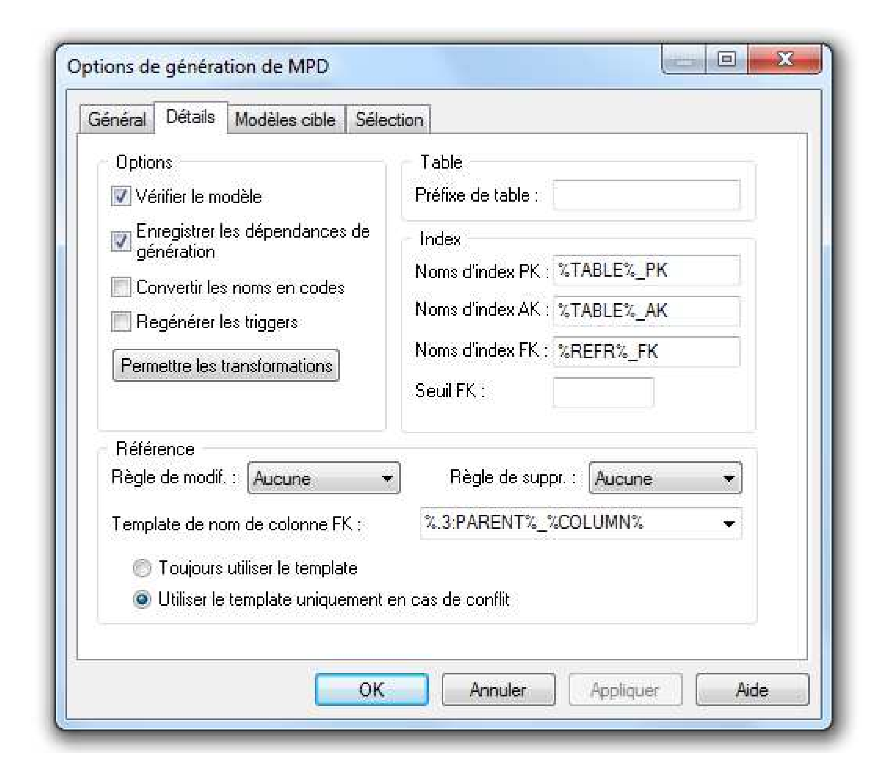
Tout comme pour le modèle logique de données, avec Power AMC, le modèle physique de données se génère automatiquement si vous avez préalablement complété comme il se doit votre Modèle Conceptuel de données. Cliquez sur « **Outils** » de la barre de menu puis « **Générer un Modèle Physique de Données…** ».



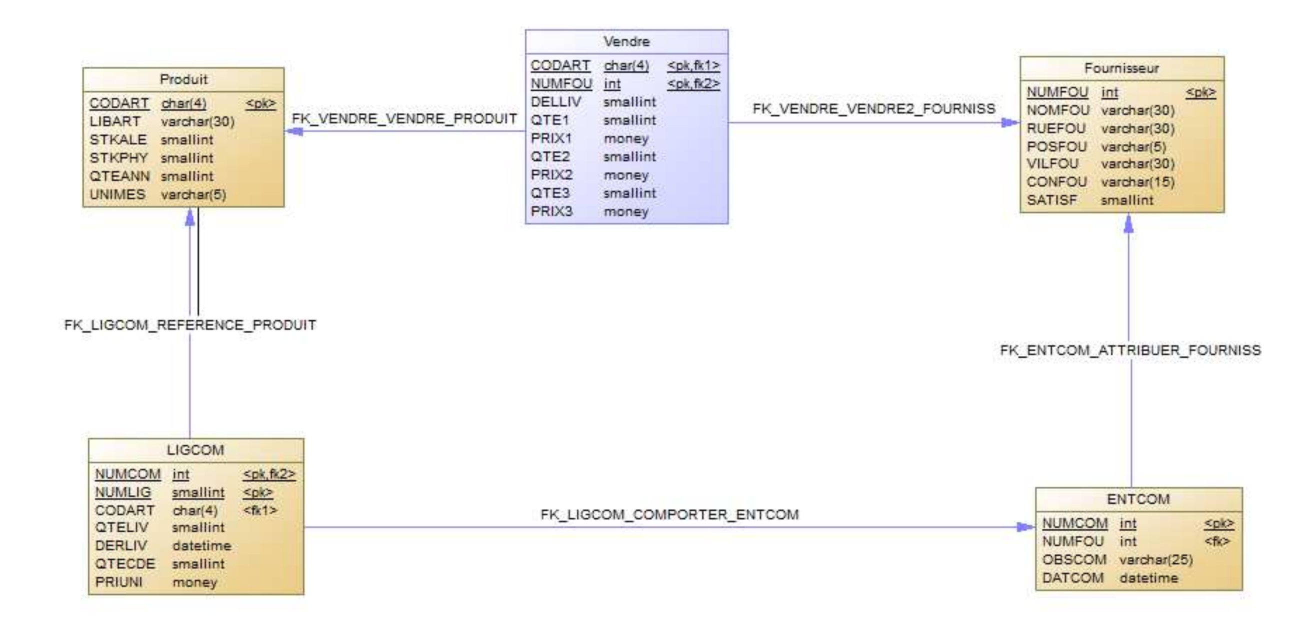
Vous devez alors configurer les différentes options et notamment le SGBD dans lequel nous allons créer notre base de données.



Sur l'onglet « **Détails** » on précisera les options suivantes :



Nous obtenons le MPD suivant :

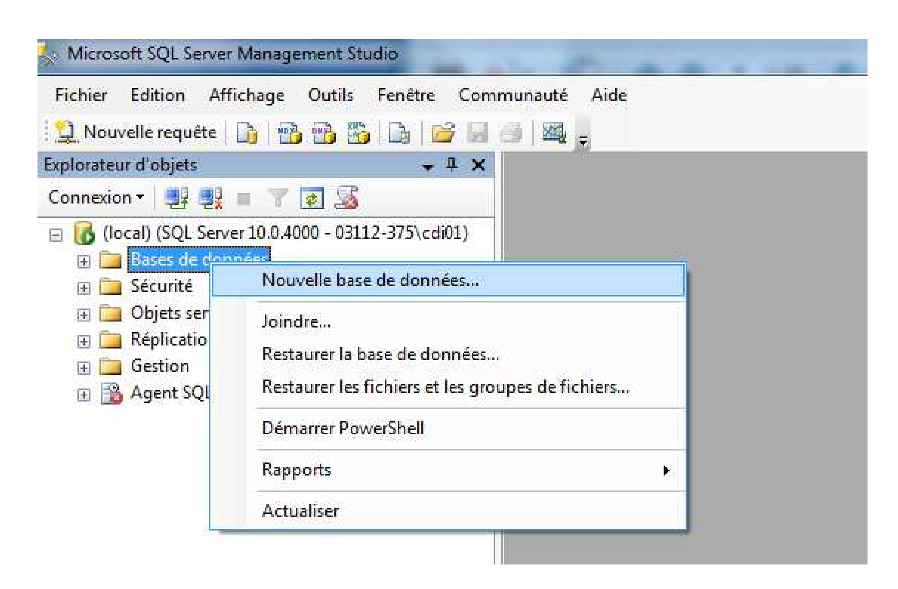


Création de la base de données sous SQL Server

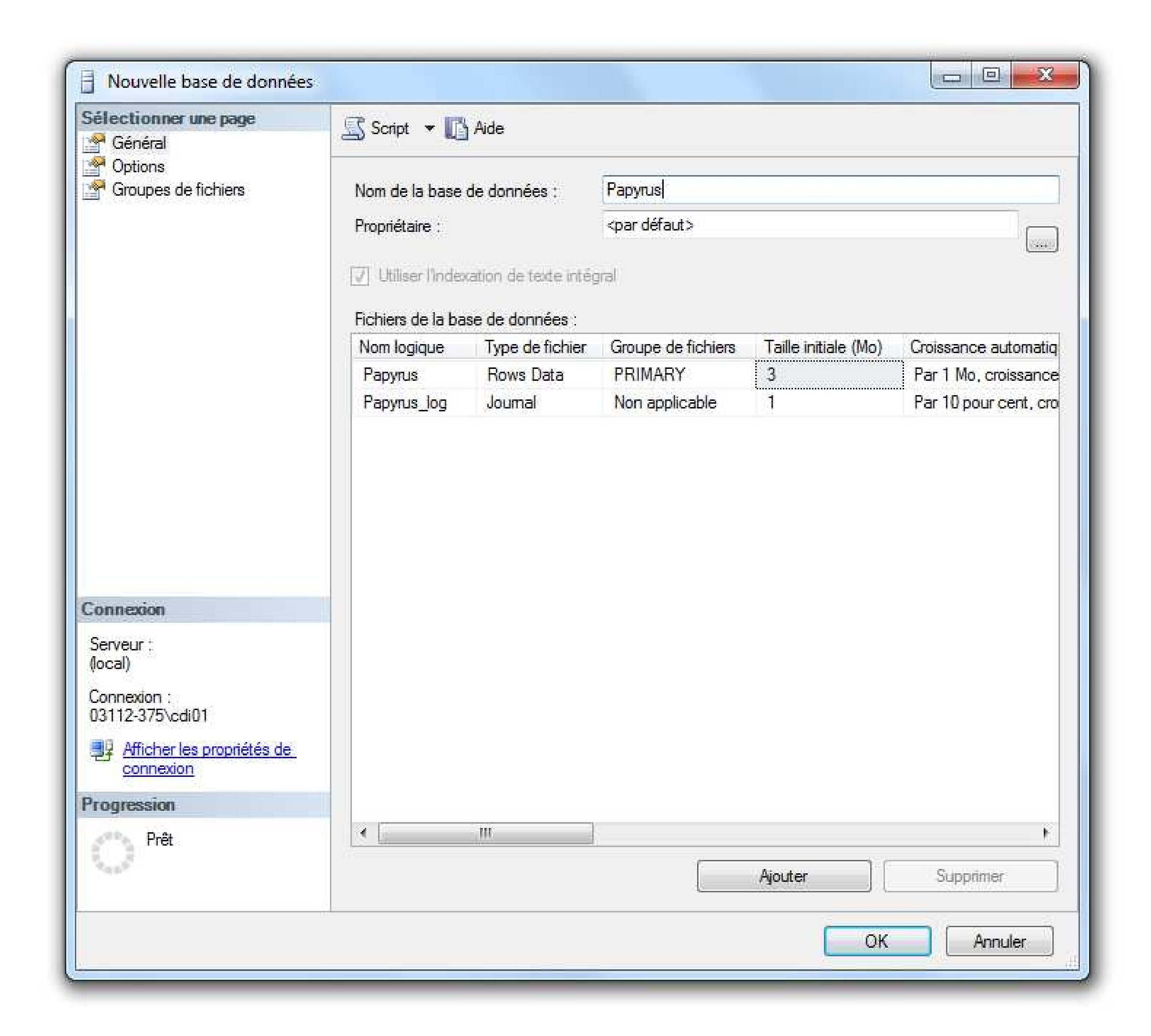
Les bases de données sont les ensembles ou nous allons stocker des objets tels que les tables, les vues, les index… Il existe deux façons de créer des bases de données sous SQL Server. En utilisant l'interface proposée par « Microsoft SQL Server Management Studio », ou bien en utilisant le code T-SQL. Chacune de ces deux méthodes possède ses avantages et ses inconvenants, il vous appartient d'adopter celle que vous trouvez la plus productive ou bien la plus pratique.

En utilisant l'interface

Pour créer une base de données sous SQL Server en utilisant le programme « Microsoft SQL Server Management Studio » (SSMS), cliquez droit sur « **Bases de données** » pour « **Nouvelle base de données…** ».



Nous nommerons la base « Papyrus ».



Par le code

Vous pouvez également créer la base de données en utilisant le langage Transact SQL. Le T-SQL (Transact Structured Query Langage) est un langage de communication avec une base de données relationnelle SQL Server. Il définit une batterie « simple », mais complète de toutes les opérations exécutables sur une base de données (lecture de données, opérations d'administration du serveur, ajout, suppression et mises à jour d'objets SQL - tables, vues, procédures stockées, déclencheurs, types de données personnalisés…). Ce langage est composé d'instructions, réparties dans de 3 catégories distinctes :

- DML : Data Modification Language, soit langage de manipulation de données. Dans cette catégorie, s'inscrivent les instructions telles que l'instruction **SELECT** ou encore les instructions qui nous permettent la création, la mise à jour et la suppression de données stockées dans les tables de la base de données. Il est important de retenir que le DML sert simplement pour les données, et en aucun cas pour la création, mise à jour ou suppression d'objets dans la base de données SQL Server.

- DDL : Data Definition Language, soit langage de définition de données. Les instructions de cette catégorie, permettent d'administrer la base de données, ainsi que les objets qu'elle contient. Elles ne permettent pas de travailler sur les données.

- **DCL :** Data Control Language, soit langage de contrôle d'accès. Cette catégorie d'instructions nous permet de gérer les accès (autorisations) aux données, aux objets SQL, aux transactions et aux configurations générales de la base.

Ces trois catégories combinées permettent que le langage T-SQL prenne en compte des fonctionnalités algorithmiques, et admette la programmabilité. Le T-SQL est non seulement un langage de requêtage, mais aussi un vrai langage de programmation à part entière. Sa capacité à écrire des procédures stockées et des déclencheurs (Triggers), lui permet d'être utilisé dans un environnement client de type .NET, au travers d'une application en C# ou en VB.NET.

```
CREATE DATABASE nomBaseDeDonnées
[ ON
[PRIMARY] [ <spécificationFichier> [,n]]
[LOG ON < spécificationFichier > [,n]]
]
[ COLLATE classement ]
[;]
```

Avec pour spécificationFichier les éléments de syntaxe suivants :

```
(NAME = nomLogique,
FILENAME = 'cheminEtNomFichier'
[,SIZE = taille [KB|MB|GB|TB]]
[,MAXSIZE={tailleMaximum[KB|MB|GB|TB]|UNLIMITED}]
[,FILEGROWTH = pasIncrement [KB|MB|GB|TB|%]]
) [,n]
```

PRIMARY

Permet de préciser le premier groupe de fichiers de la base. Ce groupe est obligatoire, car l'ensemble des tables système est obligatoirement créé dans le groupe primary. Si le mot clé **PRIMARY** est omis, le premier fichier de données précisé dans la commande **CREATE DATABASE** correspond obligatoirement au fichier primaire, il porte normalement l'extension mdf.

NAME

Cet attribut, obligatoire, permet de préciser le nom logique du fichier. Ce nom sera utilisé pour faire des manipulations sur le fichier via des commandes Transact SQL. On pense notamment aux commandes DBCC ou à la gestion de la taille des fichiers.

FILENAME

Spécification du nom et emplacement physique du fichier sur le disque dur. Le fichier de données est toujours créé sur un disque local du serveur.

SIZE

Attribut optionnel qui permet de préciser la taille du fichier de données. La taille par défaut est de 1 Mo et la taille minimale d'un fichier est 512 Ko aussi bien pour le journal que pour les fichiers de données. La taille peut être précisée en kilooctets (Kb) ou en mégaoctets (Mb, par défaut). La taille du premier fichier de données doit être au moins égale à celle de la base Model.

MAXSIZE

Cet attribut optionnel permet de préciser la taille maximale en Mégaoctets (par défaut) ou en kilooctets que peut prendre le fichier. Si rien n'est précisé, le fichier grossira jusqu'à saturation du disque dur.

FILEGROWTH

Il s'agit de préciser le facteur d'extension du fichier. La taille de chacune des extensions peut correspondre à un pourcentage (%), à une taille en Mégaoctets (par défaut) ou en Kilo octets. Si on précise la valeur zéro, le facteur d'extension automatique du fichier est nul et donc la taille du fichier ne change pas automatiquement. Si le critère **FILEGROWTH** est omis, la valeur par défaut est de 1 Mo pour les fichiers de données et de 10 % pour les fichiers journaux. La taille des extensions est toujours arrondie au multiple de 64 Ko (8 blocs) le plus proche.

Nous allons maintenant créer notre base « Papyrus » par le code. Sur « Microsoft SQL Server Management Studio » cliquez sur « **Nouvelle requête** ».



Puis saisissez votre requête.

```
/* Code TSQL */
USE [master]
GO
/****** Object: Database [Papyrus] Script Date: 17/01/2011 15:59:20 By
Stéphane Grare ******/
CREATE DATABASE [Papyrus] ON PRIMARY
( NAME = N'Test',
FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\Papyrus.mdf' ,
SIZE = 3072KB ,
MAXSIZE = UNLIMITED,
FILEGROWTH = 1024KB )
LOG ON
( NAME = N'Test_log',
FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\Papyrus_log.ldf' ,
SIZE = 1024KB ,
MAXSIZE = 2048GB ,
FILEGROWTH = 10%)
GO
```

Une fois votre requête saisie, cliquez sur le bouton ! Exécuter pour lancer la requête. Si votre requête c'est bien exécuté, un message : « Commande(s) réussie(s) » s'affiche alors.

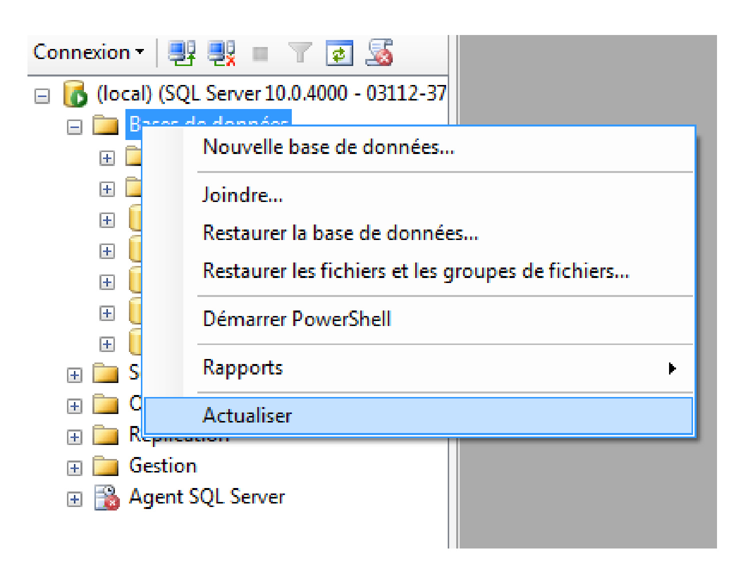
SQLQuery5.sql - (l...2-375\cdi01 (52))*

```sql
/* Code TSQL */
USE [master]
GO
/****** Object: Database [Papyrus] Script Date: 17/01/2011 15:59:20 By Stéphane Grare ******/
CREATE DATABASE [Papyrus] ON PRIMARY
( NAME = N'Test',
FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\Papyrus.mdf' ,
SIZE = 3072KB ,
MAXSIZE = UNLIMITED,
FILEGROWTH = 1024KB )
LOG ON
( NAME = N'Test_log',
FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\Papyrus_log.ldf' ,
SIZE = 1024KB ,
MAXSIZE = 2048GB ,
FILEGROWTH = 10%)
GO
```

Messages

Commande(s) réussie(s).

Vous devrez alors cliquer droit, « **Actualiser** » pour afficher votre nouvelle base.



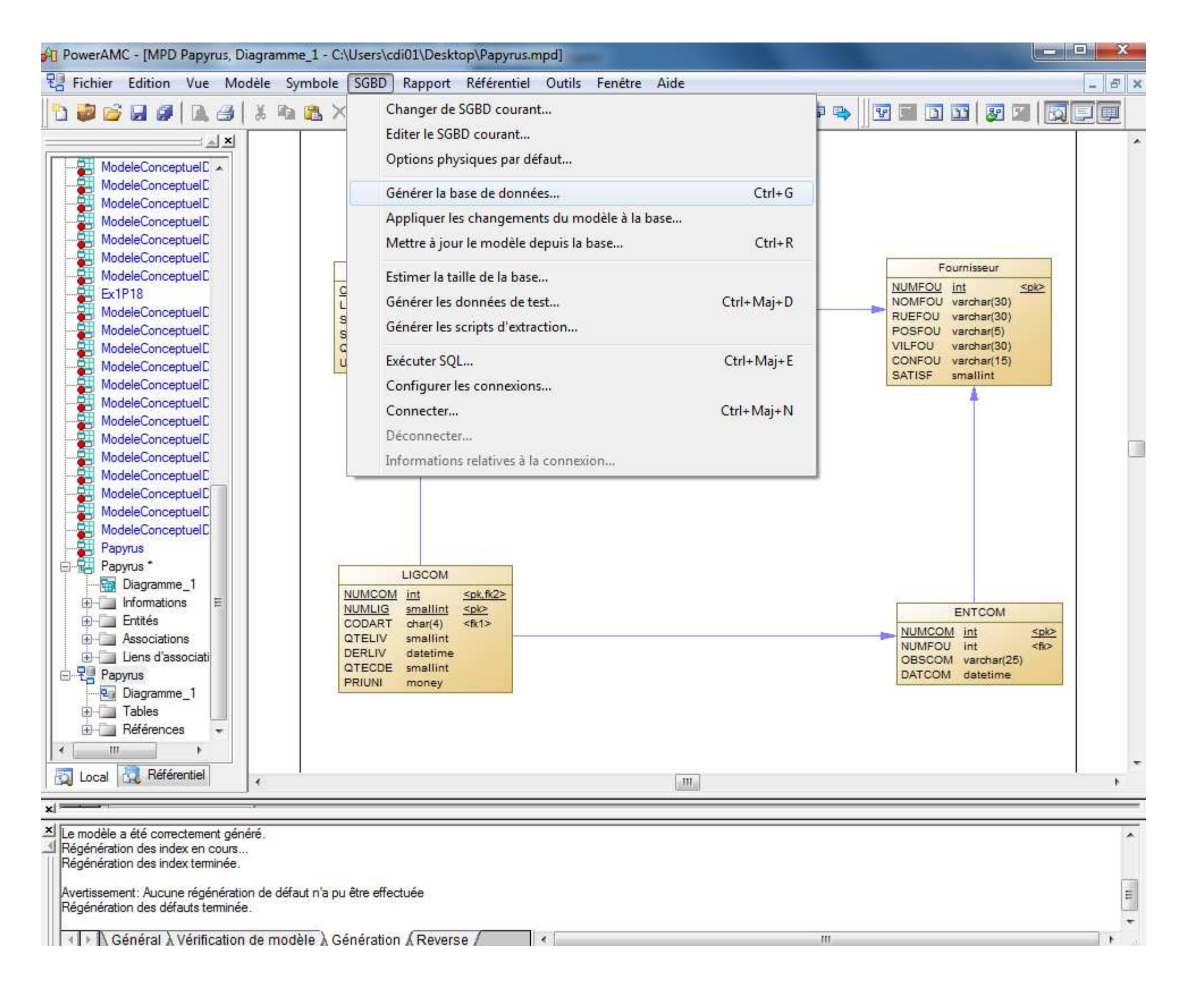
Création de tables sous SQL Server

Nous avons 3 possibilités pour créer les tables qui composeront notre base de données.

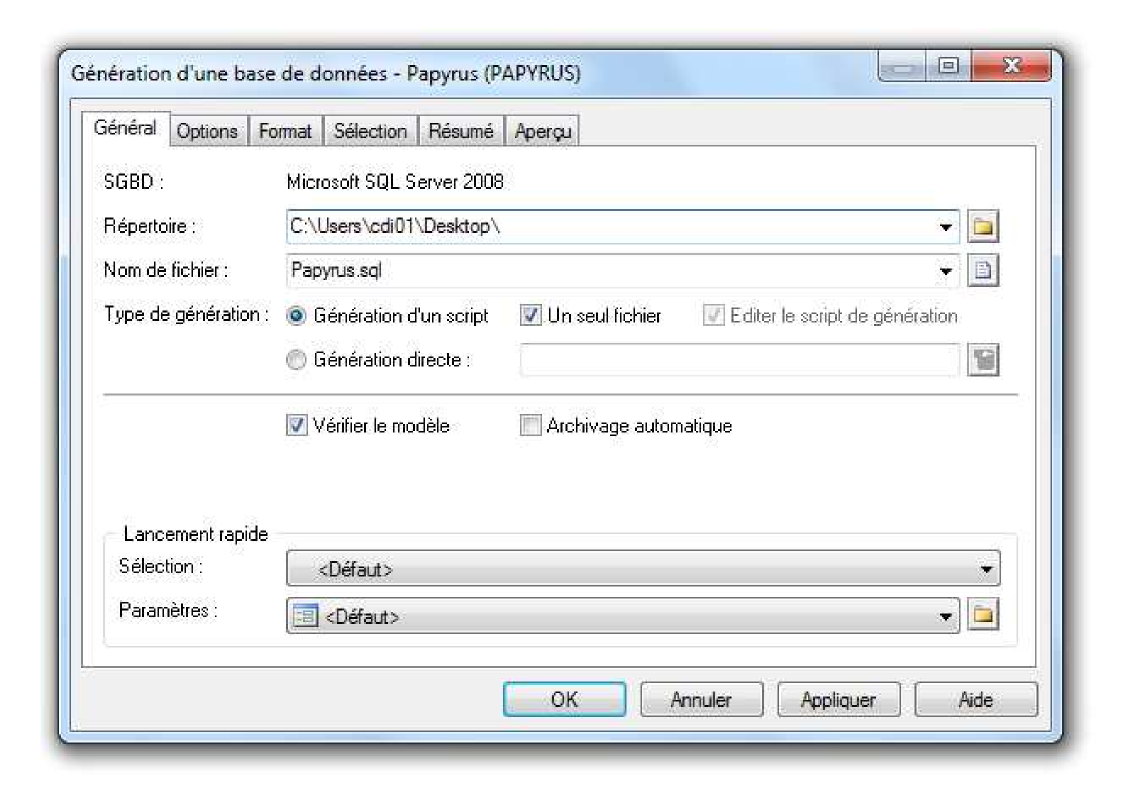
- Avec Power AMC
- Manuellement
- Par le code

Avec Power AMC

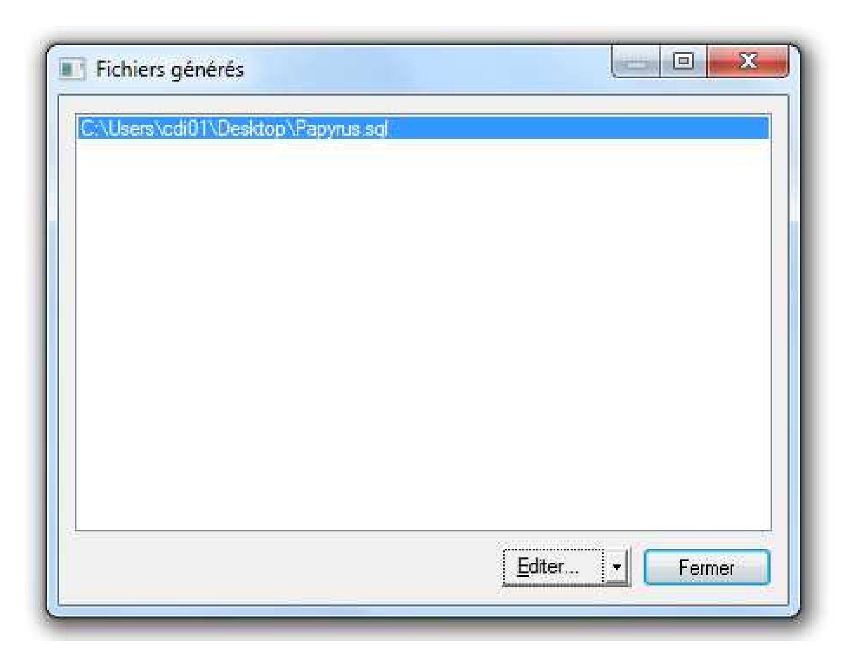
Sur Power AMC, à partir de notre modèle physique de données générer précédemment, cliquer sur « **SGBD** » de la barre de menu de Power AMC puis « **Générer la base de données** ».



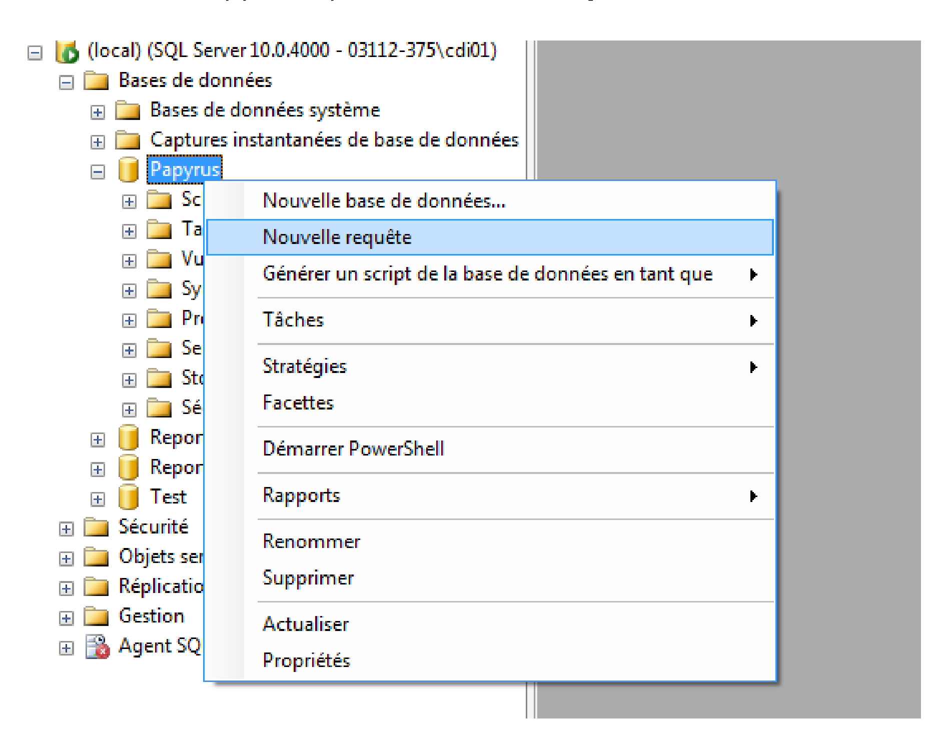
Sélectionnez le répertoire de sortie et le nom du fichier *.sql. Nous le nommerons « Papyrus.sql ». Vous avez également la possibilité de configurer différentes options.



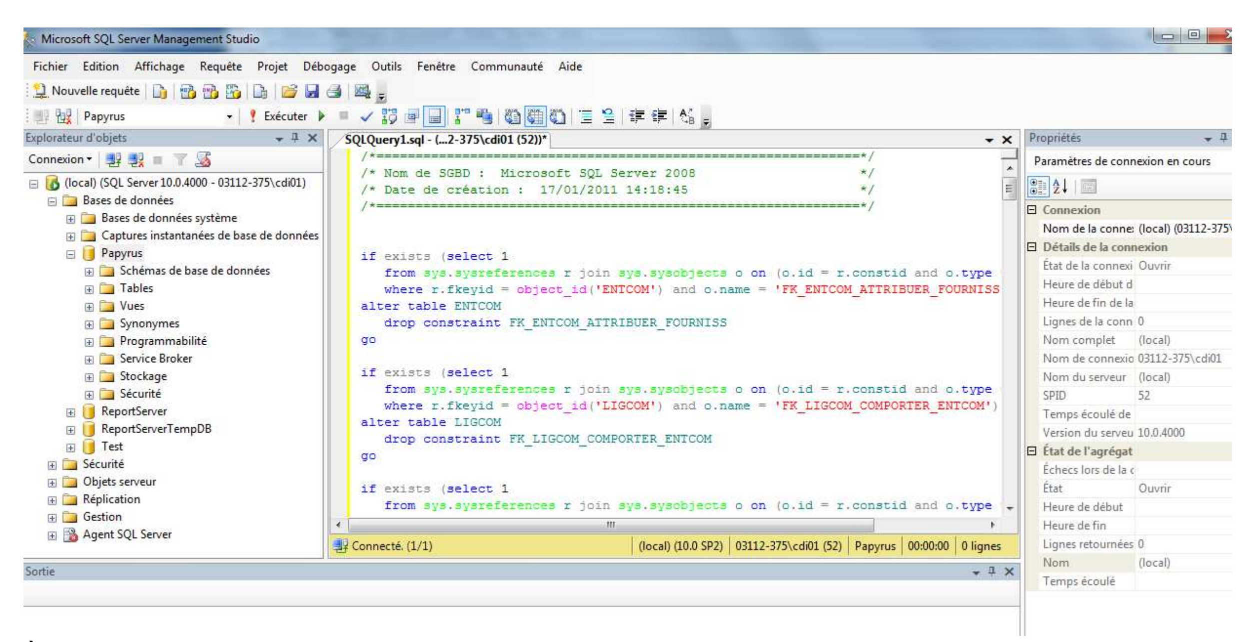
Le fichier est alors généré à l'emplacement indiquer.



Revenons sur le programme « Microsoft SQL Server Management Studio ». Cliquez droit sur votre base de données « Papyrus » puis « **Nouvelle requête** ».



Copier / coller le contenu du fichier généré par Power AMC que nous avons nommé « Papyrus.sql ». On peut également passer par le menu Fichier / Ouvrir / Fichiers…



À ce stade, on peut modifier le code SQL afin d'ajouter des contraintes, de modifier les types… Nous verrons plus loin qu'il existe d'autres possibilités.

```
/*==============================================================*/
/* Nom de SGBD :  Microsoft SQL Server 2008                     */
/* Date de création :  17/01/2011 14:29:07                      */
/*==============================================================*/


if exists (select 1
   from sys.sysreferences r join sys.sysobjects o on (o.id = r.constid and
o.type = 'F')
   where r.fkeyid = object_id('ENTCOM') and o.name =
'FK_ENTCOM_ATTRIBUER_FOURNISS')
alter table ENTCOM
   drop constraint FK_ENTCOM_ATTRIBUER_FOURNISS
go

if exists (select 1
   from sys.sysreferences r join sys.sysobjects o on (o.id = r.constid and
o.type = 'F')
   where r.fkeyid = object_id('LIGCOM') and o.name =
'FK_LIGCOM_COMPORTER_ENTCOM')
alter table LIGCOM
   drop constraint FK_LIGCOM_COMPORTER_ENTCOM
go

if exists (select 1
   from sys.sysreferences r join sys.sysobjects o on (o.id = r.constid and
o.type = 'F')
   where r.fkeyid = object_id('LIGCOM') and o.name =
'FK_LIGCOM_REFERENCE_PRODUIT')
alter table LIGCOM
   drop constraint FK_LIGCOM_REFERENCE_PRODUIT
go

if exists (select 1
   from sys.sysreferences r join sys.sysobjects o on (o.id = r.constid and
o.type = 'F')
   where r.fkeyid = object_id('VENDRE') and o.name =
'FK_VENDRE_VENDRE_PRODUIT')
```

```
alter table VENDRE
   drop constraint FK_VENDRE_VENDRE_PRODUIT
go

if exists (select 1
   from sys.sysreferences r join sys.sysobjects o on (o.id = r.constid and 
o.type = 'F')
   where r.fkeyid = object_id('VENDRE') and o.name = 
'FK_VENDRE_VENDRE2_FOURNISS')
alter table VENDRE
   drop constraint FK_VENDRE_VENDRE2_FOURNISS
go

if exists (select 1
            from  sysindexes
           where  id    = object_id('ENTCOM')
            and   name  = 'ATTRIBUER_FK'
            and   indid > 0
            and   indid < 255)
   drop index ENTCOM.ATTRIBUER_FK
go

if exists (select 1
            from  sysobjects
           where  id = object_id('ENTCOM')
            and   type = 'U')
   drop table ENTCOM
go

if exists (select 1
            from  sysobjects
           where  id = object_id('FOURNISSEUR')
            and   type = 'U')
   drop table FOURNISSEUR
go

if exists (select 1
            from  sysindexes
           where  id    = object_id('LIGCOM')
            and   name  = 'COMPORTER_FK'
            and   indid > 0
            and   indid < 255)
   drop index LIGCOM.COMPORTER_FK
go

if exists (select 1
            from  sysindexes
           where  id    = object_id('LIGCOM')
            and   name  = 'REFERENCER_FK'
            and   indid > 0
            and   indid < 255)
   drop index LIGCOM.REFERENCER_FK
go

if exists (select 1
            from  sysobjects
           where  id = object_id('LIGCOM')
            and   type = 'U')
   drop table LIGCOM
go
```

```sql
if exists (select 1
            from  sysobjects
           where  id = object_id('PRODUIT')
            and   type = 'U')
   drop table PRODUIT
go

if exists (select 1
            from  sysindexes
           where  id    = object_id('VENDRE')
            and   name  = 'VENDRE2_FK'
            and   indid > 0
            and   indid < 255)
   drop index VENDRE.VENDRE2_FK
go

if exists (select 1
            from  sysindexes
           where  id    = object_id('VENDRE')
            and   name  = 'VENDRE_FK'
            and   indid > 0
            and   indid < 255)
   drop index VENDRE.VENDRE_FK
go

if exists (select 1
            from  sysobjects
           where  id = object_id('VENDRE')
            and   type = 'U')
   drop table VENDRE
go

/*==============================================================*/
/* Table : ENTCOM                                               */
/*==============================================================*/
create table ENTCOM (
   NUMCOM               int                  not null,
   NUMFOU               int                  not null,
   OBSCOM               varchar(25)          not null,
   DATCOM               datetime             not null,
   constraint PK_ENTCOM primary key nonclustered (NUMCOM)
)
go

/*==============================================================*/
/* Index : ATTRIBUER_FK                                         */
/*==============================================================*/
create index ATTRIBUER_FK on ENTCOM (
NUMFOU ASC
)
go

/*==============================================================*/
/* Table : FOURNISSEUR                                          */
/*==============================================================*/
create table FOURNISSEUR (
   NUMFOU               int                  not null,
   NOMFOU               varchar(30)          not null,
   RUEFOU               varchar(30)          not null,
   POSFOU               varchar(5)           not null,
   VILFOU               varchar(30)          not null,
```

```
   CONFOU               varchar(15)          not null,
   SATISF               smallint             not null,
   constraint PK_FOURNISSEUR primary key nonclustered (NUMFOU)
)
go

/*==============================================================*/
/* Table : LIGCOM                                               */
/*==============================================================*/
create table LIGCOM (
   NUMCOM               int                  not null,
   NUMLIG               smallint             not null,
   CODART               char(4)              not null,
   QTELIV               smallint             not null,
   DERLIV               datetime             not null,
   QTECDE               smallint             not null,
   PRIUNI               money                not null,
   constraint PK_LIGCOM primary key nonclustered (NUMCOM, NUMLIG)
)
go

/*==============================================================*/
/* Index : REFERENCER_FK                                        */
/*==============================================================*/
create index REFERENCER_FK on LIGCOM (
CODART ASC
)
go

/*==============================================================*/
/* Index : COMPORTER_FK                                         */
/*==============================================================*/
create index COMPORTER_FK on LIGCOM (
NUMCOM ASC
)
go

/*==============================================================*/
/* Table : PRODUIT                                              */
/*==============================================================*/
create table PRODUIT (
   CODART               char(4)              not null,
   LIBART               varchar(30)          not null,
   STKALE               smallint             not null,
   STKPHY               smallint             not null,
   QTEANN               smallint             not null,
   UNIMES               varchar(5)           not null,
   constraint PK_PRODUIT primary key nonclustered (CODART)
)
go

/*==============================================================*/
/* Table : VENDRE                                               */
/*==============================================================*/
create table VENDRE (
   CODART               char(4)              not null,
   NUMFOU               int                  not null,
   DELLIV               smallint             not null,
   QTE1                 smallint             not null,
   PRIX1                money                not null,
   QTE2                 smallint             not null,
```

```
   PRIX2                money                not null,
   QTE3                 smallint             not null,
   PRIX3                money                null,
   constraint PK_VENDRE primary key (CODART, NUMFOU)
)
go

/*==============================================================*/
/* Index : VENDRE_FK                                            */
/*==============================================================*/
create index VENDRE_FK on VENDRE (
CODART ASC
)
go

/*==============================================================*/
/* Index : VENDRE2_FK                                           */
/*==============================================================*/
create index VENDRE2_FK on VENDRE (
NUMFOU ASC
)
go

alter table ENTCOM
   add constraint FK_ENTCOM_ATTRIBUER_FOURNISS foreign key (NUMFOU)
      references FOURNISSEUR (NUMFOU)
go

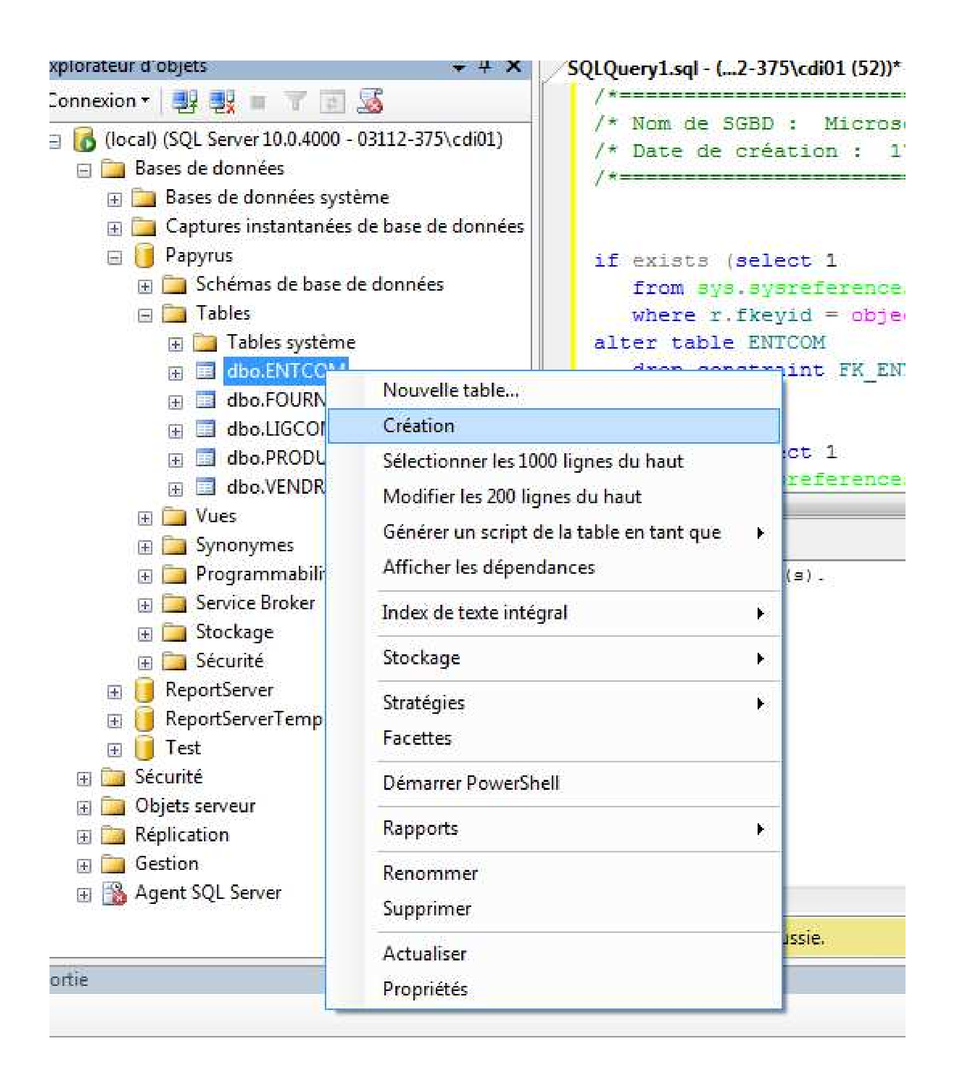
alter table LIGCOM
   add constraint FK_LIGCOM_COMPORTER_ENTCOM foreign key (NUMCOM)
      references ENTCOM (NUMCOM)
go

alter table LIGCOM
   add constraint FK_LIGCOM_REFERENCE_PRODUIT foreign key (CODART)
      references PRODUIT (CODART)
go

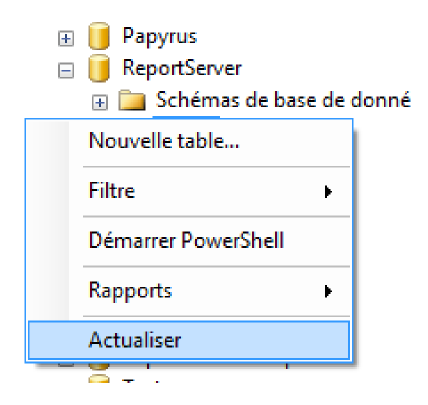
alter table VENDRE
   add constraint FK_VENDRE_VENDRE_PRODUIT foreign key (CODART)
      references PRODUIT (CODART)
go

alter table VENDRE
   add constraint FK_VENDRE_VENDRE2_FOURNISS foreign key (NUMFOU)
      references FOURNISSEUR (NUMFOU)
go
```

Pour exécuter votre requête, cliquez sur le bouton ! Exécuter pour lancer la requête. Si votre requête c'est bien exécuté, un message : « Commande(s) réussie(s) » s'affiche alors.

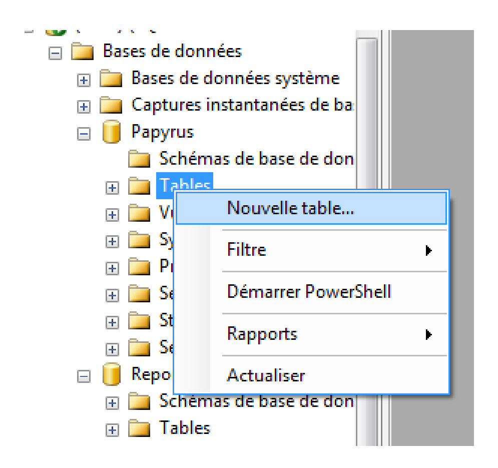


Cliquez droit pour actualiser vos données.

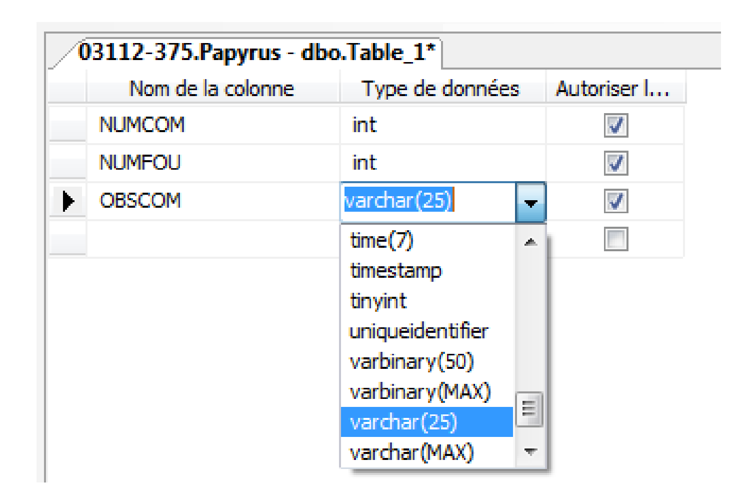


Par l'interface

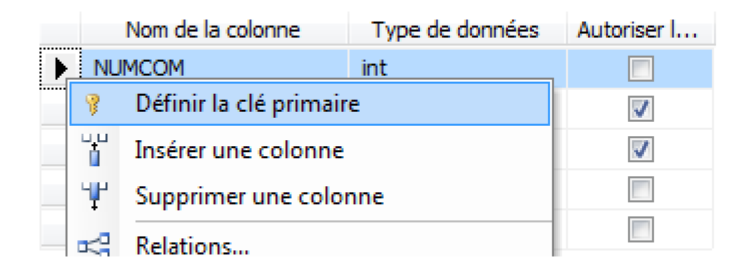
Vous pouvez créer vos tables manuellement dans « Microsoft SQL Server Management Studio ». Sur la base de données que vous venez de créer, sélectionnez le dossier table puis cliquez droit / Nouvelle table…



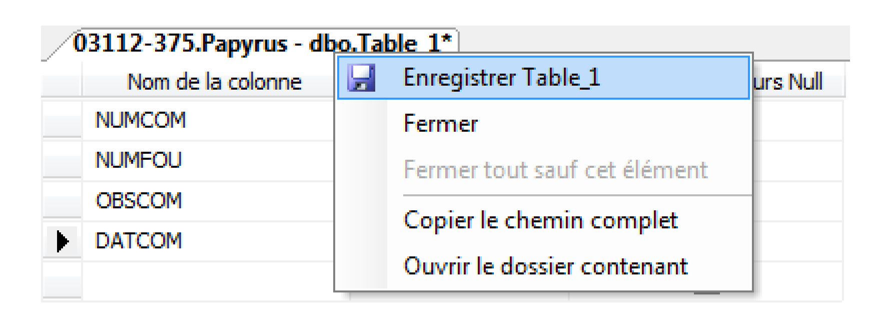
Indiquez le nom de la colonne, le type de données à l'aide du menu déroulant et décochez la case si vous ne souhaitez pas autoriser les valeurs null, c'est-à-dire l'absence d'une donnée dans la colonne.



N'oubliez pas de définir la clé primaire de vos tables. Cliquez droit / Définir la clé primaire.



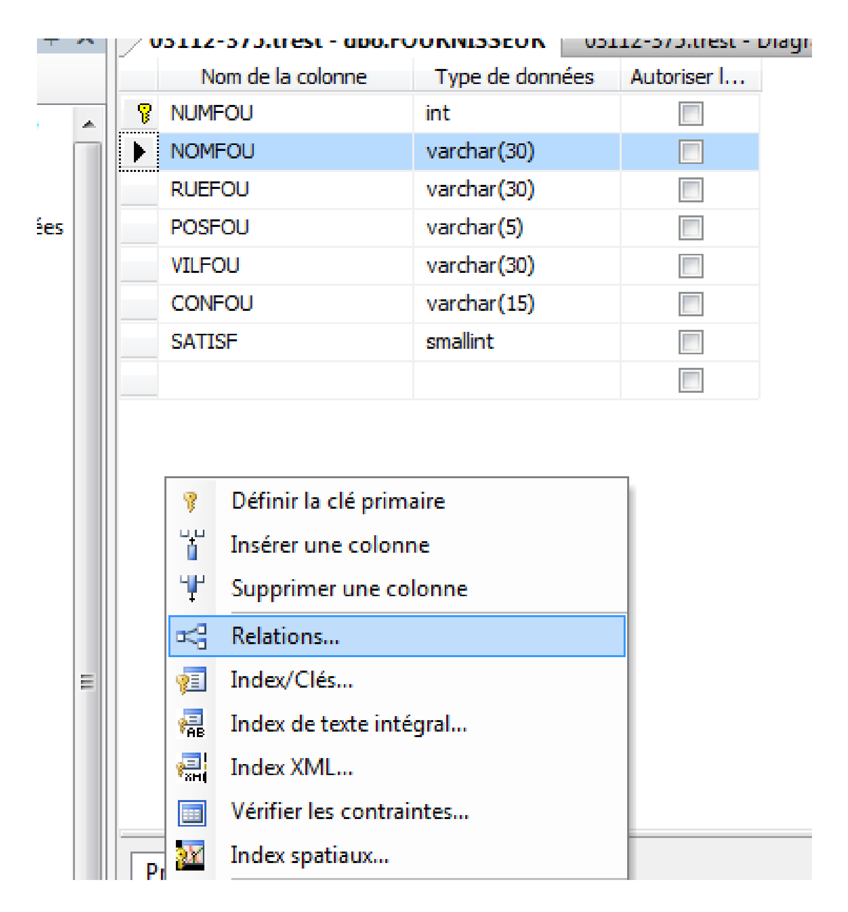
Une fois créer, vous pouvez l'enregistrer en cliquant droit Enregistrer Table.



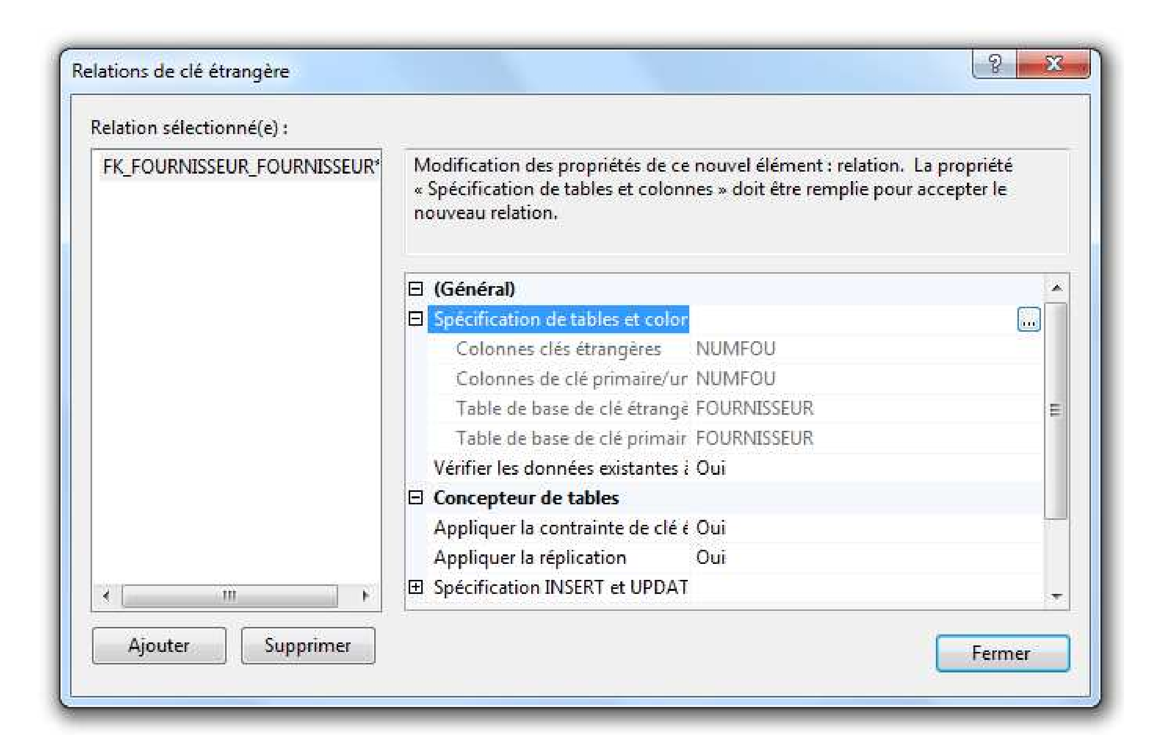
Vous devrez alors donner un nom à votre table.



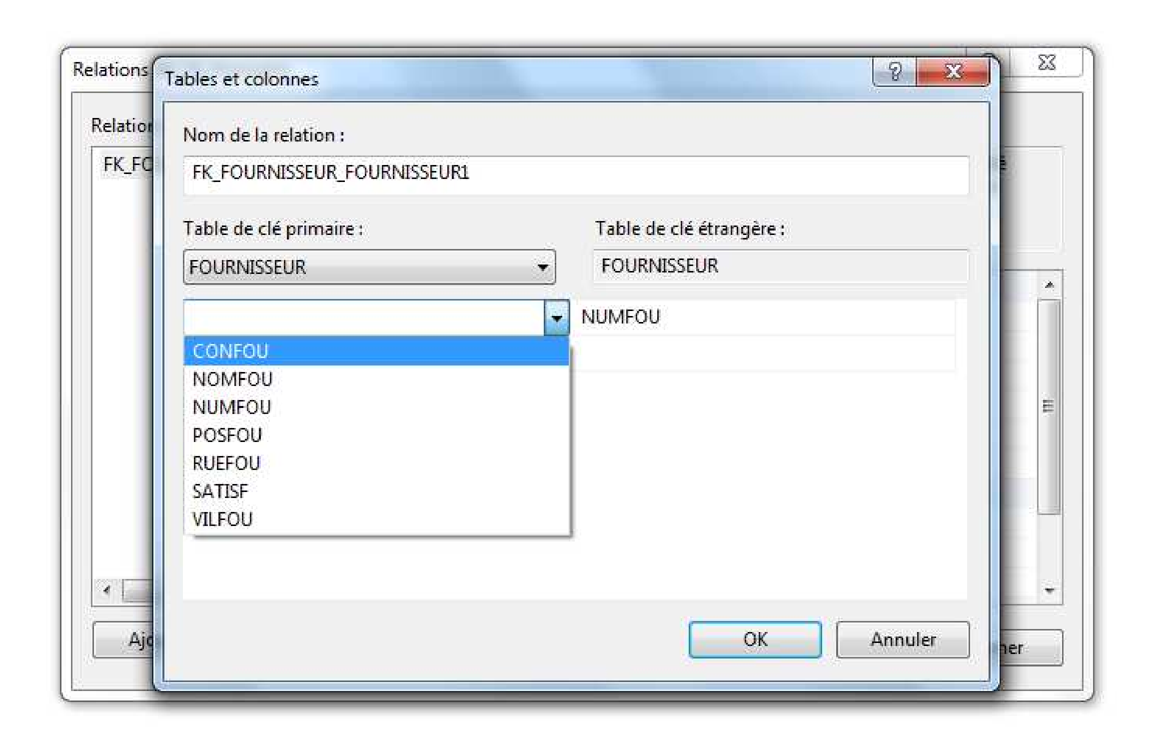
Pour définir la clé étrangère, cliquez droit puis « **Relations** ».



Puis cliquez sur **ajouter** pour créer un relation. La relation s'affiche dans la liste Relation sélectionnée avec un nom fourni par système au format FK_<tablename>_<tablename>, où tablename est le nom de la table de clé étrangère. Sélectionnez-la puis dans « **Spécification de tables et color** » cliquez sur ....

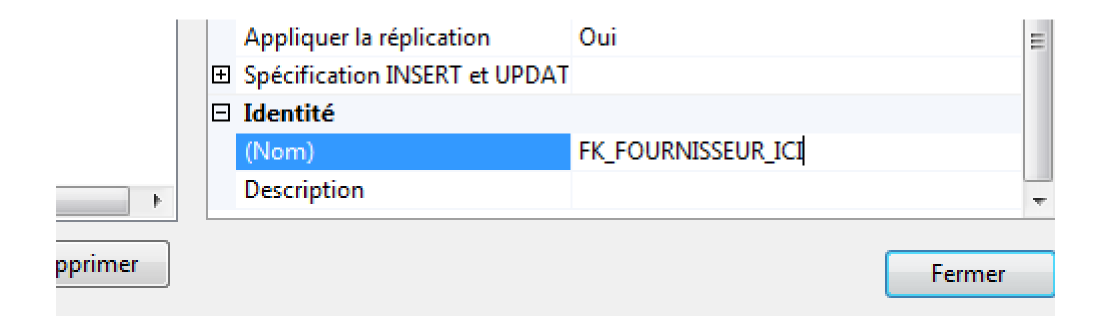


Dans la liste déroulante Clé primaire de la boîte de dialogue Tables et Colonnes, choisissez la table qui sera du côté clé primaire de la relation.



Dans la grille située au-dessous, choisissez les colonnes qui participent à la clé primaire de la table. Dans la cellule de la grille située à gauche de chaque colonne, choisissez la colonne de clé étrangère correspondante dans la table de clé étrangère.

Le Concepteur de tables propose un nom pour la relation. Pour changer ce nom, modifiez le contenu de la zone de texte à Identité. Cliquez sur OK pour créer la relation.



Les colonnes que vous choisissez pour la clé étrangère doivent avoir le même type de données que les colonnes primaires correspondantes. Chacune des clés doit comprendre un nombre égal de colonnes. Par exemple, si la clé primaire de la table du côté clé primaire de la relation est composée de deux colonnes, vous devez faire correspondre chacune de ces colonnes à une colonne de la table pour le côté clé étrangère de la relation.

Par le code

Nous pouvons également créer des tables en utilisant le langage Transact SQL. L'étape de création des tables est une étape importante de la conception de la base, car les données sont organisées par rapport aux tables.

```
CREATE TABLE [nomSchema.] nom_table
( nom_colonne {typecolonne|AS expression_calculée}
[,nom_colonne ... ][,contraintes...])
[ON groupefichier]
[TEXTIMAGE_ON groupe_fichier]
```

nomSchema

Nom du schéma dans lequel la table va être définie.

nom_table

Peut-être sous la forme base.propriétaire.table.

nom_colonne

Nom de la colonne qui doit être unique dans la table.

typecolonne

Type système ou type défini par l'utilisateur.

contraintes

Règles d'intégrité.

Groupefichier (*filegroup)*

Groupe de fichiers sur lequel va être créée la table.

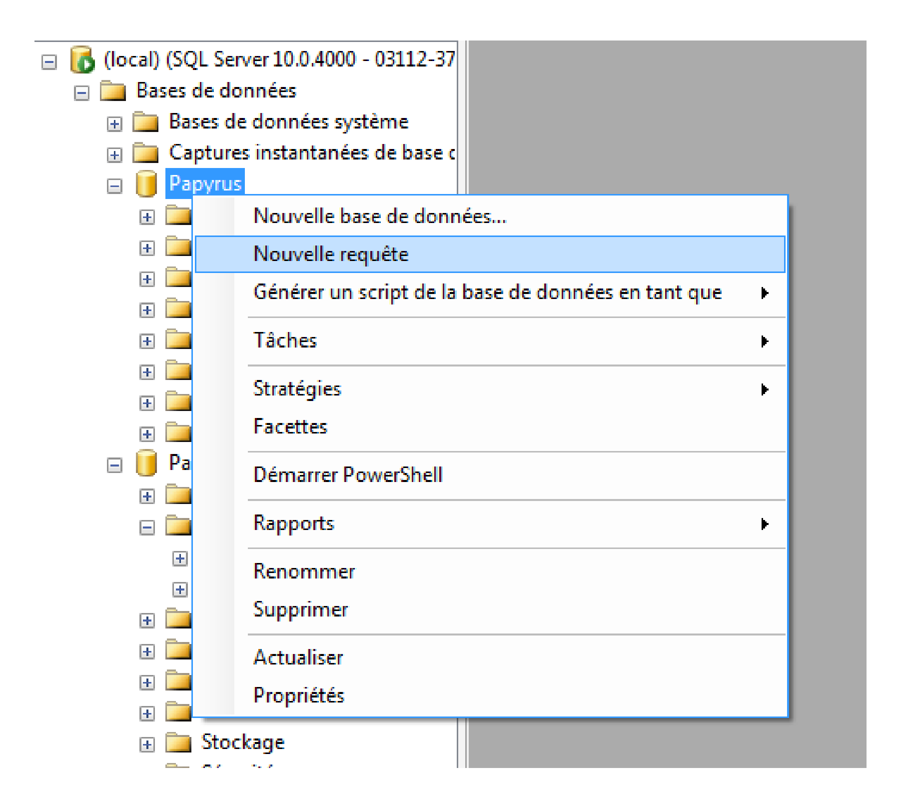
AS expression_calculée

Il est possible de définir une règle de calcul pour les colonnes qui contiennent des données calculées. Bien entendu, ces colonnes ne sont accessibles qu'en lecture seule, et il n'est pas possible d'insérer des données ou de mettre à jour les données d'une telle colonne.

TEXTIMAGE_ON

Permet de préciser le groupe de fichiers destination pour les données de type texte et image.
Il est possible de créer 2 milliards de tables par base de données. Le nombre maximal de colonnes par table est de 1024. La longueur maximale d'une ligne est de 8060 octets (sans compter les données texte ou image).

Revenons à notre cas Papyrus. Cliquez droit sur votre base de données puis « **Nouvelle requête**… ».



Saisissons notre code Transact SQL suivant :

```
create table FOURNISSEUR (
   NUMFOU               int                  not null,
   NOMFOU               varchar(30)          not null,
   RUEFOU               varchar(30)          not null,
   POSFOU               varchar(5)           not null,
   VILFOU               varchar(30)          not null,
   CONFOU               varchar(15)          not null,
   SATISF               smallint             not null,
   constraint PK_FOURNISSEUR primary key nonclustered (NUMFOU)
)
go
```

Pour exécuter votre requête, cliquez sur le bouton Exécuter pour lancer la requête. Si votre requête c'est bien exécuté, un message : « Commande(s) réussie(s) » s'affiche alors.

Une contrainte d'intégrité est une clause permettant de contraindre la modification de tables, faite par l'intermédiaire de requêtes d'utilisateurs, afin que les données saisies dans la base soient conformes aux données attendues. Ces contraintes doivent être exprimées dès la création de la table grâce aux mots clés suivants :

- CONSTRAINT
- DEFAULT
- NOT NULL
- UNIQUE
- CHECK
- CONSTRAINT
- DEFAULT
- NOT NULL
- UNIQUE
- CHECK

Le langage SQL permet de définir une valeur par défaut lorsqu'un champ de la base n'est pas renseigné grâce à la clause **DEFAULT**. Cela permet notamment de faciliter la création de tables, ainsi que de garantir qu'un champ ne sera pas vide.

La clause **DEFAULT** doit être suivie par la valeur à affecter. Cette valeur peut être un des types suivants : Constante numérique, constante alphanumérique (chaîne de caractères), le mot clé **USER** (nom de l'utilisateur), le mot clé **NULL**, le mot clé **CURRENT_DATE** (date de saisie), le mot clé **CURRENT_TIME** (heure de saisie), le mot clé **CURRENT_TIMESTAMP** (date et heure de saisie).

Le mot clé **NOT NULL** permet de spécifier qu'un champ doit être saisi, c'est-à-dire que le SGBD refusera d'insérer des tuples dont un champ comportant la clause **NOT NULL** n'est pas renseigné.

Il est possible de faire un test sur un champ grâce à la clause **CHECK()** comportant une condition logique portant sur une valeur entre les parenthèses. Si la valeur saisie est différente de **NULL**, le SGBD va effectuer un test grâce à la condition logique. Celui-ci peut éventuellement être une condition avec des ordres **SELECT**...

La clause **UNIQUE** permet de vérifier que la valeur saisie pour un champ n'existe pas déjà dans la table. Cela permet de garantir que toutes les valeurs d'une colonne d'une table seront différentes.

```
CREATE TABLE MATABLE1
(COLONNE1 int UNIQUE)
```

La clause **IDENTITY** peut être affectée à une colonne par table, de type numérique entier. Elle permet d'incrémenter les valeurs d'une colonne, ligne après ligne. Par défaut, la contrainte **IDENTITY** part de 1, et a un pas d'incrément de 1. Il est possible de changer la valeur de départ et le pas d'incrément. Proposons un script qui crée une table, avec deux colonnes, une de type **IDENTITY** et une avec un type char, et faisons plusieurs insertions dans cette table.

```
-- Créez la table avec la contrainte IDENTITY
CREATE TABLE MATABLE
(COLONNE1 NUMERIC(18,0) IDENTITY,
COLONNE2 char(10))
```

```
-- Insertion multiple dans notre nouvelle table
INSERT INTO MATABLE
(COLONNE2)
VALUES
('Cours 1'),('Cours 2'),('Cours 3')
```

Remarquez que lorsque l'on insère des lignes dans une table comportant une colonne **IDENTITY**, nous n'avons pas besoin de préciser la colonne et la valeur qu'elle prend en argument, d'où son intérêt, d'automatiser la saisie des données. Vérifions maintenant le résultat avec un simple **SELECT** :

DELTA1.Test - dbo.MATABLE

	COLONNE1	COLONNE2
▶	1	Cours 1
	2	Cours 2
	3	Cours 3
*	NULL	NULL

On remarque bien que la colonne COLONNE1 s'est peuplée seule, grâce à la contrainte **IDENTITY**. Il est bon de rappeler que nous n'avons droit qu'à une seule contrainte **IDENTITY** par table.

Il est possible de donner un nom à une contrainte grâce au mot clé **CONSTRAINT** suivi du nom que l'on donne à la contrainte, de telle manière à ce que le nom donné s'affiche en cas de non-respect de l'intégrité, c'est-à-dire lorsque la clause que l'on a spécifiée n'est pas validée.

Si la clause **CONSTRAINT** n'est pas spécifiée, un nom sera donné arbitrairement par le SGBD. Toutefois, le nom donné par le SGBD risque fortement de ne pas être compréhensible, et ne sera vraisemblablement pas compris lorsqu'il y aura une erreur d'intégrité. La stipulation de cette clause est donc fortement conseillée.

Exemple : Voici un exemple permettant de voir la syntaxe d'une instruction de création de tables avec contraintes :

```
CREATE TABLE clients(
Nom char(30) NOT NULL,
Prenom char(30) NOT NULL,
Age integer, check (age < 100),
Email char(50) NOT NULL, check (Email LIKE "%@%")
)
```

Grâce à SQL, il est possible de définir des clés, c'est-à-dire spécifier la (ou les) colonne(s) dont la connaissance permet de désigner précisément un et un seul tuple (une ligne).

L'ensemble des colonnes faisant partie de la table en cours permettant de désigner de façon unique un tuple est appelé clé primaire et se définit grâce à la clause **PRIMARY KEY** suivie de la liste de colonnes, séparées par des virgules, entre parenthèses. Ces colonnes ne peuvent alors plus prendre la valeur **NULL** et doivent être telles que deux lignes ne puissent avoir simultanément la même combinaison de valeurs pour ces colonnes.

```
PRIMARY KEY (colonne1, colonne2, ...)
```

Lorsqu'une liste de colonnes de la table en cours de définition permet de définir la clé primaire d'une table étrangère, on parle alors de clé étrangère, et on utilise la clause **FOREIGN KEY** suivie de la liste de colonnes de la table en cours de définition, séparée par des virgules, entre parenthèses, puis de la clause **REFERENCES** suivie du nom de la table étrangère et de la liste de ses colonnes correspondantes, séparées par des virgules, entre parenthèses.

```
FOREIGN KEY (colonne1, colonne2, ...)
REFERENCES Nom_de_la_table_etrangere(colonne1,colonne2,...)
Trigger (gâchette): -- Garantie de l'intégrité référentielle
```

Les clés étrangères permettent de définir les colonnes d'une table garantissant la validité d'une autre table. Ainsi, il existe des éléments (appelés triggers, ou en français gâchettes ou déclencheurs) permettant de garantir l'ensemble de ces contraintes que l'on désigne par le terme d'intégrité référentielle, c'est-à-dire notamment de s'assurer qu'un tuple utilisé à partir d'une autre table existe réellement.

Ces triggers sont **ON DELETE** et **ON UPDATE** :

ON DELETE est suivi d'arguments entre accolades permettant de spécifier l'action à réaliser en cas d'effacement d'une ligne de la table faisant partie de la clé étrangère :

- **CASCADE** indique la suppression en cascade des lignes de la table étrangère dont les clés étrangères correspondent aux clés primaires des lignes effacées.
- **RESTRICT** indique une erreur en cas d'effacement d'une valeur correspondant à la clé.
- **SET NULL** place la valeur **NULL** dans la ligne de la table étrangère en cas d'effacement d'une valeur correspondant à la clé.
- **SET DEFAULT** place la valeur par défaut (qui suit ce paramètre) dans la ligne de la table étrangère en cas d'effacement d'une valeur correspondant à la clé.

ON UPDATE est suivi d'arguments entre accolades permettant de spécifier l'action à réaliser en cas de modification d'une ligne de la table faisant partie de la clé étrangère :

- **CASCADE** indique la modification en cascade des lignes de la table étrangère dont les clés primaires correspondent aux clés étrangères des lignes modifiées.
- **RESTRICT** indique une erreur en cas de modification d'une valeur correspondant à la clé.
- **SET NULL** place la valeur **NULL** dans la ligne de la table étrangère en cas de modification d'une valeur correspondant à la clé.
- **SET DEFAULT** place la valeur par défaut (qui suit ce paramètre) dans la ligne de la table étrangère en cas de modification d'une valeur correspondant à la clé.

Exemple : On créer une base test dans laquelle on exécute le script suivant. Ont créé deux tables avec les contraintes suivantes.

```
USE Test
CREATE TABLE MATABLE1
(COLONNE1 int PRIMARY KEY)
CREATE TABLE MATABLE2
(COLONNE1 int CONSTRAINT FK_MATABLE2 FOREIGN KEY (COLONNE1)
REFERENCES MATABLE1 (COLONNE1)
ON DELETE CASCADE)
```

On enregistre les données suivantes :

DELTA1.Test - dbo.MATABLE1

	COLONNE1
	10
	11
	12
	13
	14
▶*	NULL

DELTA1.Test - dbo.MATABLE2

	COLONNE1
	10
	11
	12
	13
	14
▶*	NULL

Maintenant, on supprime la ligne 2 (11) de la table MATABLE1 et on actualise notre base de données puis on affiche de nouveau les deux tables : on constate que la mise à jour a été automatiquement faite dans la table 2.

DELTA1.Test - dbo.MATABLE1

	COLONNE1
▶	10
	12
	13
	14
*	NULL

DELTA1.Test - dbo.MATABLE2

	COLONNE1
▶	10
	14
	12
	13
*	NULL

Les assertions sont des expressions devant être satisfaites lors de la modification de données pour que celles-ci puissent être réalisées. Ainsi, elles permettent de garantir l'intégrité des données. Leur syntaxe est la suivante :

```
CREATE ASSERTION Nom_de_la_contrainte CHECK (expression_conditionnelle)
```

La condition à remplir peut (et est généralement) être effectuée grâce à une clause **SELECT**.

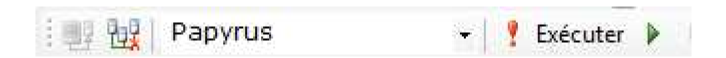
Pour notre base de données « Papyrus », si votre table contient une clé étrangère :

```
use Papyrus
create table VENDRE (
   CODART                 char(4)              not null,
   NUMFOU                 int                  not null,
   DELLIV                 smallint             not null,
   QTE1                   smallint             not null,
   PRIX1                  money                not null,
   QTE2                   smallint             not null,
   PRIX2                  money                not null,
```

```
   QTE3                  smallint              not null,
   PRIX3                 money                 null,
   constraint PK_VENDRE primary key (CODART, NUMFOU),
   constraint FK_VENDRE_VENDRE_PRODUIT foreign key (CODART)
   references PRODUIT (CODART)
)
go
```

La clé primaire est le champ CODART ET NUMFOU. La contrainte **FOREIGN KEY**, basé sur la colonne CODART référence le champ CODART de la table PRODUIT pour garantir que toute vente est associée à un produit existant.

L'instruction **USE** vous permet de spécifier le fichier de base de données SQL Server à interroger (spécifier la base de données concernée par la requête) lors de l'utilisation de SQL Server Management Studio.



Si vous n'utilisez pas **USE**, assurez-vous lorsque vous effectué une requête que celle-ci s'applique bien à la base de données sur laquelle vous travaillez.

NB : Pour faire des commentaires dans vos requêtes…

```
--Commentaires sur une seule ligne

/* Commentaires
sur
plusieurs
lignes */
```

La contrainte **CHECK** permet de vérifier, avant insertion ou mise à jour des données contenues dans la colonne en question, que les données à insérer sont bien au format voulu, ou encore qu'une valeur entrée dans la colonne pour un enregistrement appartiendra à un domaine de valeurs particulier. Regardons maintenant la syntaxe de cette contrainte :

```
CREATE TABLE MATABLE1
(COLONNE1 int CHECK (expression_booleenne))
```

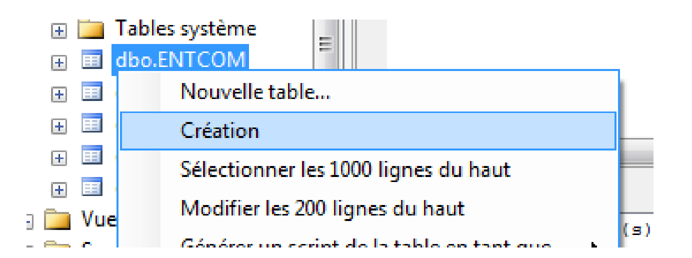
Il est possible d'ajouter l'option **NOT FOR REPLICATION** après le mot clé **CHECK**, afin de spécifier qu'il faut empêcher l'application de la contrainte dans un cas de réplication.

Modifications de tables et contraintes

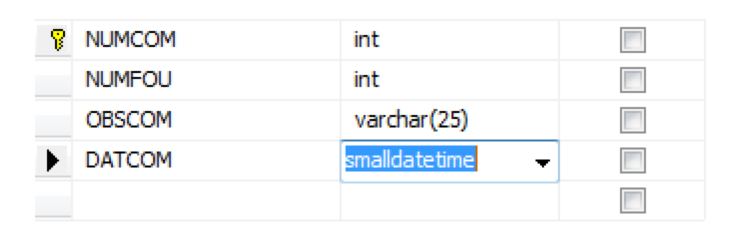
Pour modifier une table sous SQL Server, vous avez deux possibilités : en utilisant l'interface de « Microsoft SQL Server Management Studio » ou par le code.

En utilisant l'interface

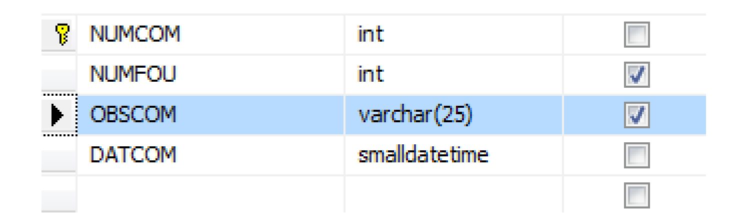
Nous devons modifier manuellement les types, car dans Power AMC, il n'y a pas tous les types que propose SQL Server. Pour modifier une table manuellement, cliquez droit sur la table à modifier puis « **Création** ».



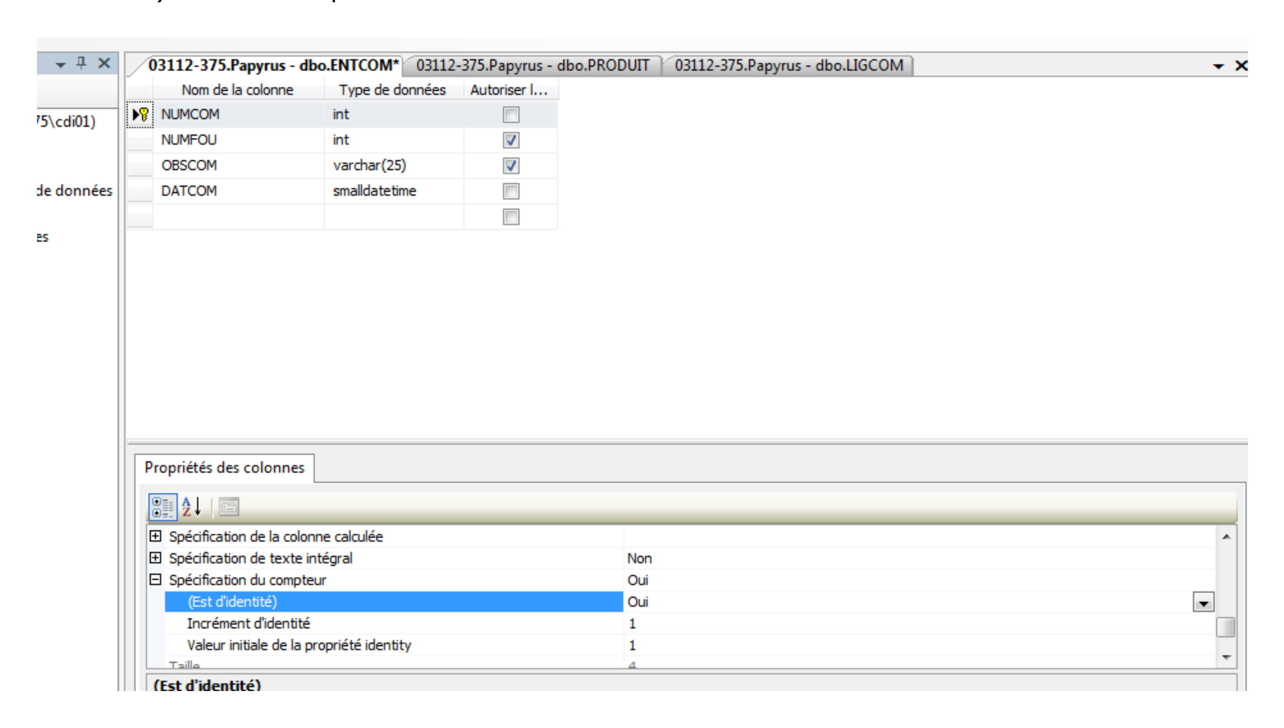
Vous pouvez alors modifier les données à votre guise.



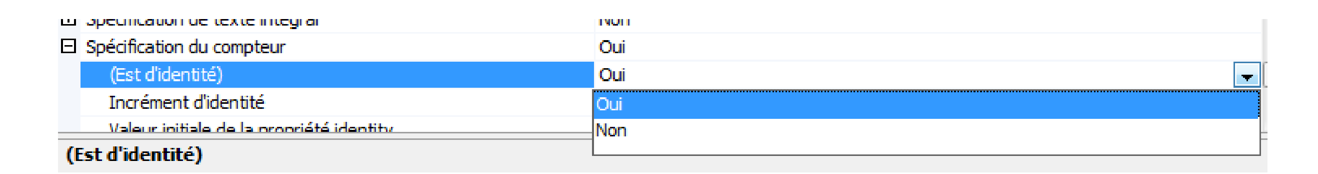
À noter qu'avec SQL Server, vous avez également la possibilité de préciser si on peut autoriser les types null (données absentes) en cochant les cases.



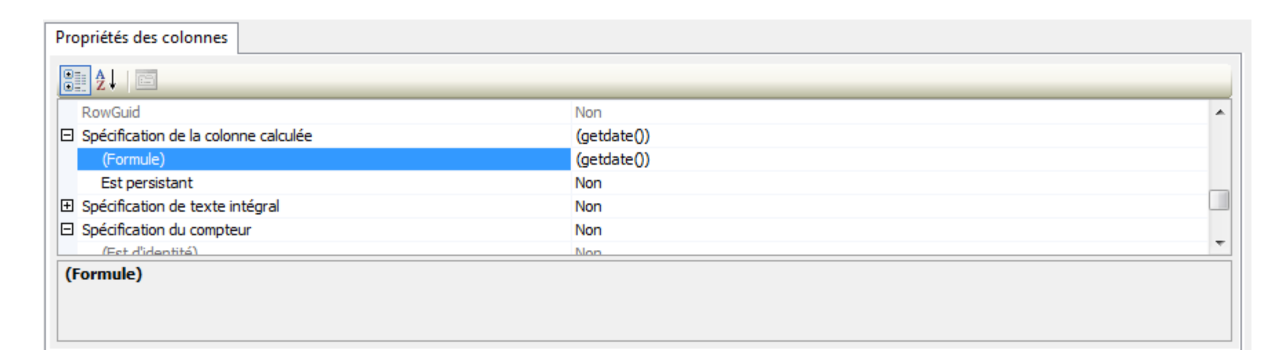
Définissons également les contraintes répondant à l'énoncé du problème : « Le numéro de commande est un champ compteur auto incrémenté de 1 ». On indiquera « **oui** » à (est d'identité) sur-le-champ NUMCOM.



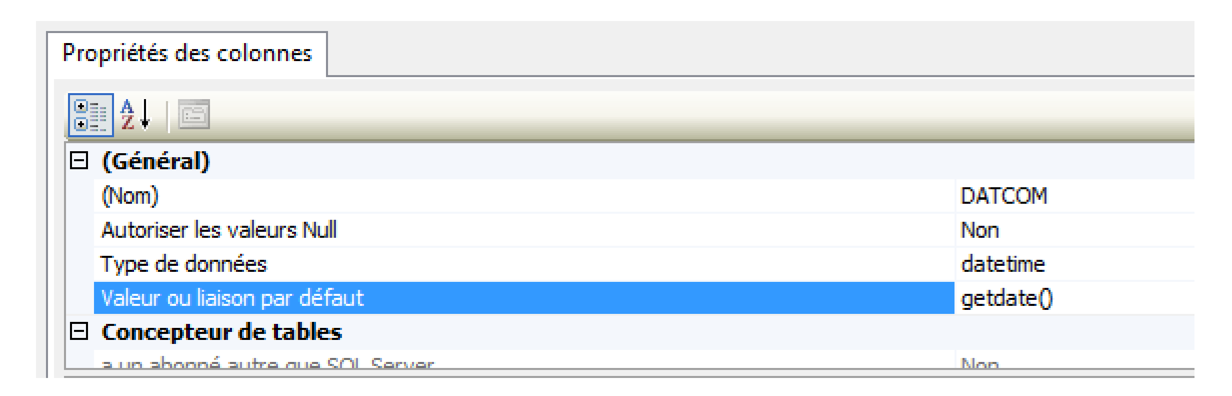
Il suffit d'utiliser le menu déroulant et de sélectionner « **Oui** ».



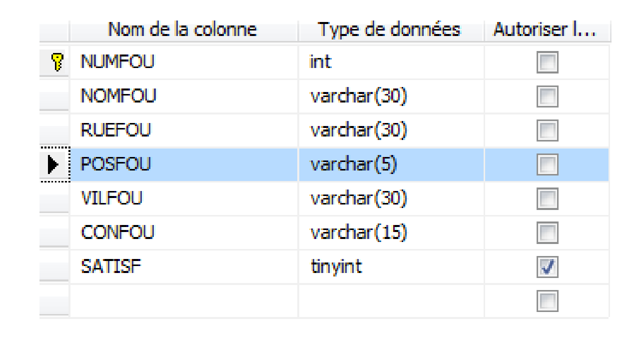
« La date de commande est par défaut la date du jour ». Pour la colonne DATCOM on utilisera la formule **getdate()**.

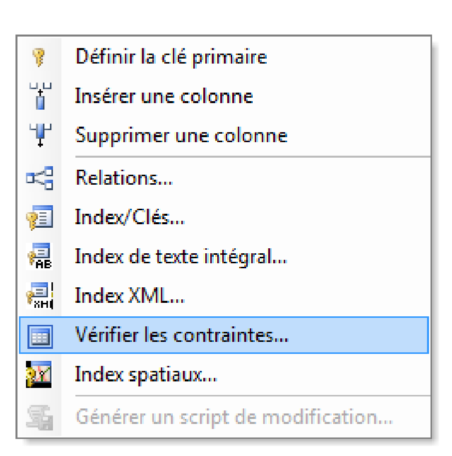


Le problème, c'est que nous avons contraint notre colonne à la date du jour. Il n'est par conséquent pas possible de saisir une autre date que la date du jour. On supprime donc la formule précédente. Remontez en haut des propriétés de la colonne et inscrivez **getdate()** à « Valeur ou liaison par défaut ». La colonne proposera la date du jour, mais celle-ci sera alors modifiable.

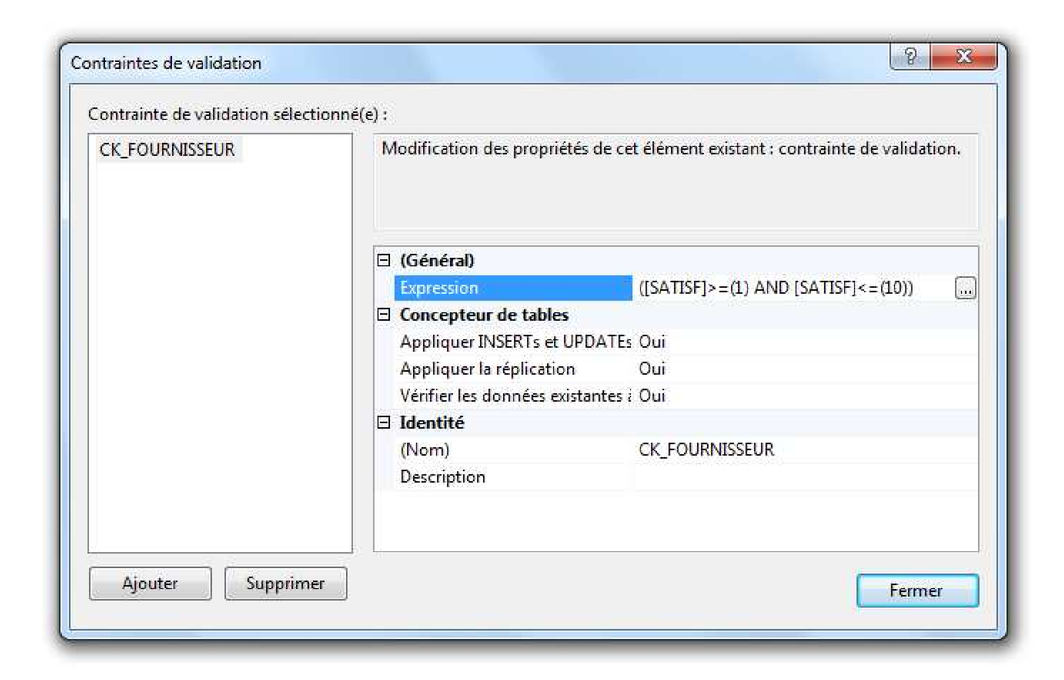


« L'indice de satisfaction est compris dans une échelle de 1 à 10 ». Cliquez droit dans la table correspondante puis **« Vérifier les contraintes** ».

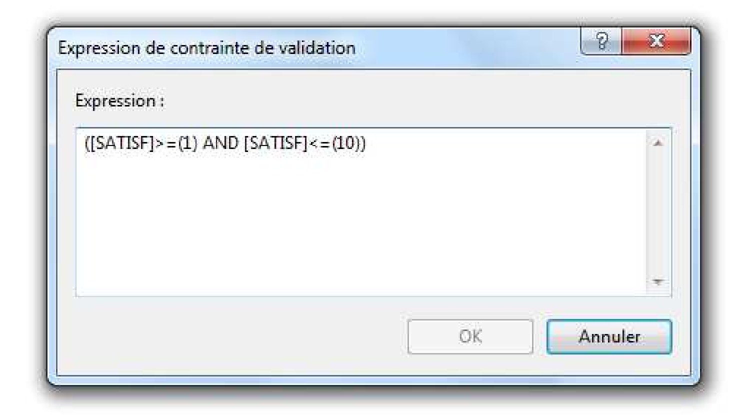




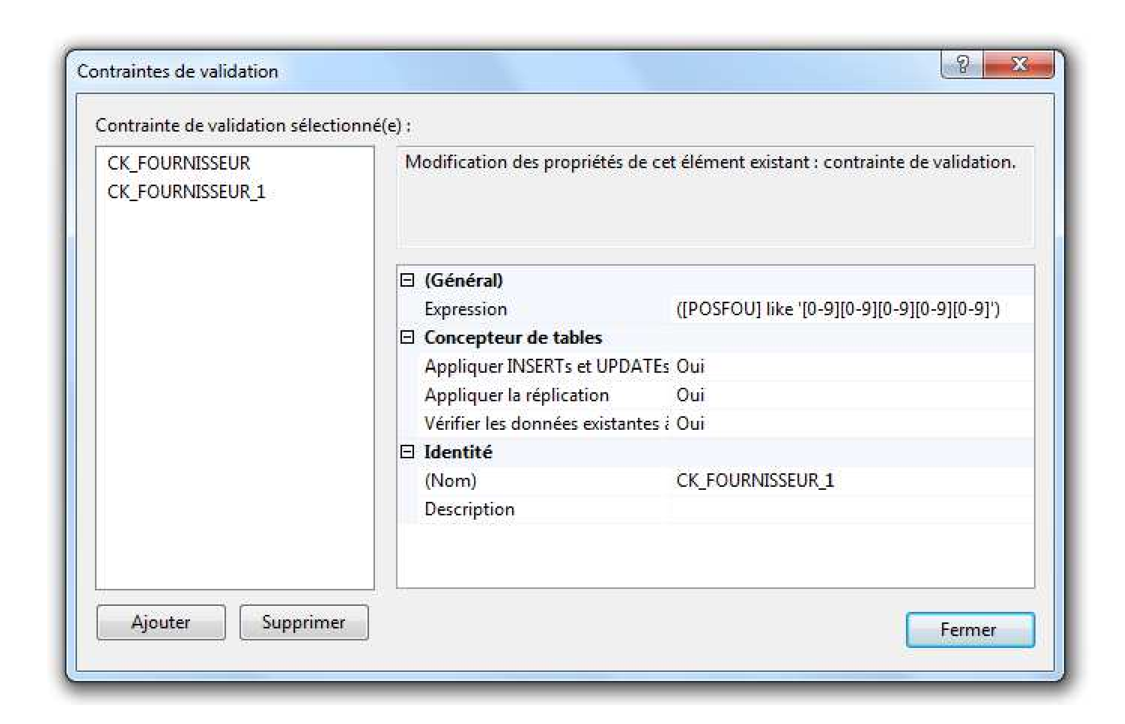
Cliquez sur « Ajouter » pour ajouter une contrainte. Dans expression, saisissez votre contrainte : **[SATISF]>=(1) AND [SATISF]<=(10)**



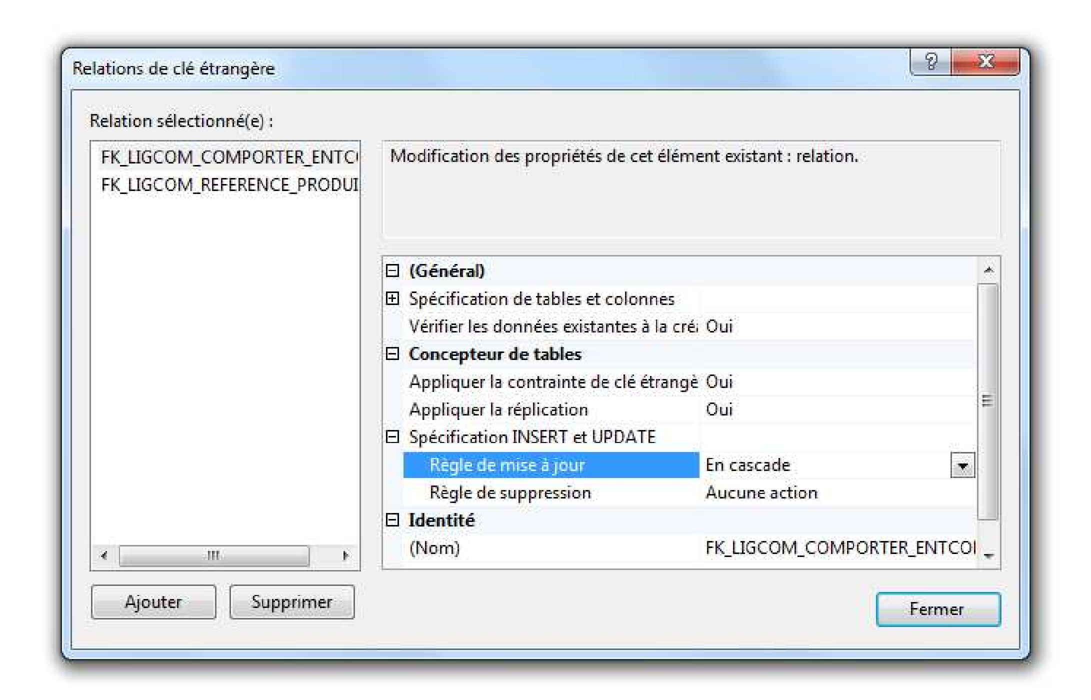
Les parenthèses s'ajoutent automatiquement.



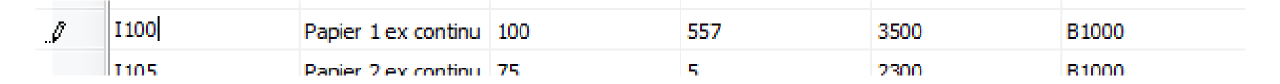
« Le code postal est constitué de 5 chiffres ». De la même manière, on ajoute une deuxième contrainte. Nous saisirons : **[POSFOU] like '[0-9][0-9][0-9][0-9][0-9]'**



Il existe aussi des contraintes avec des règles de mises à jour ou de suppression en cas de spécification d'un **INSERT** ou d'un **UPDATE**. Pour mieux comprendre cette notion, nous allons prendre un exemple concret. Nous allons modifier les contraintes des tables LIGCOM, PRODUIT et VENDRE de façon à ce que lorsque nous modifierons le code d'un produit de la table PRODUIT (à l'aide d'une instruction **UPDATE** par le langage SQL), la mise à jour se reporte automatiquement sur les tables LIGCOM et VENDRE. Pour cela nous spécifions la règle de mise à jour « **En cascade** » comme ci-dessous sur l'écran des Relations de clé étrangère.

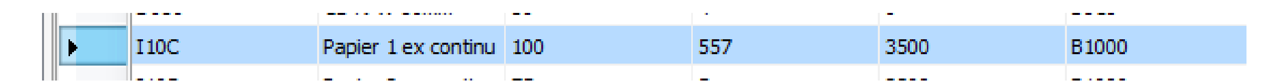


Nous procédons donc de cette façon pour les tables LIGCOM, VENDRE et PRODUIT. Une fois les relations de chacune des tables modifiées, nous allons mettre à jour notre table PRODUIT. Mettons à jour le code de l'article « I100 » par exemple.



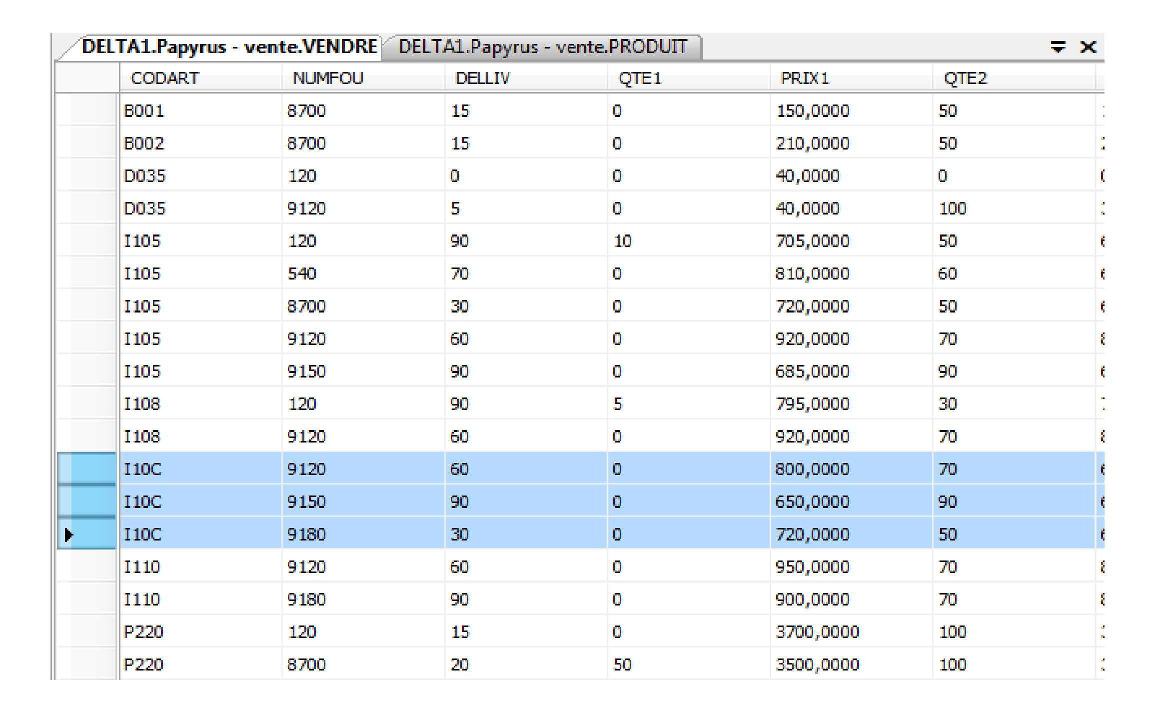
I100	Papier 1 ex continu	100	557	3500	B1000
I105	Papier 2 ex continu	75	5	2300	B1000

Nous changeons le code de l'article en « I10C ». On enregistre les modifications et on actualise notre base de données.



I10C	Papier 1 ex continu	100	557	3500	B1000

On interroge maintenant la table VENDRE. On constate que celle-ci a été modifiée automatiquement.



DELTA1.Papyrus - vente.VENDRE | DELTA1.Papyrus - vente.PRODUIT

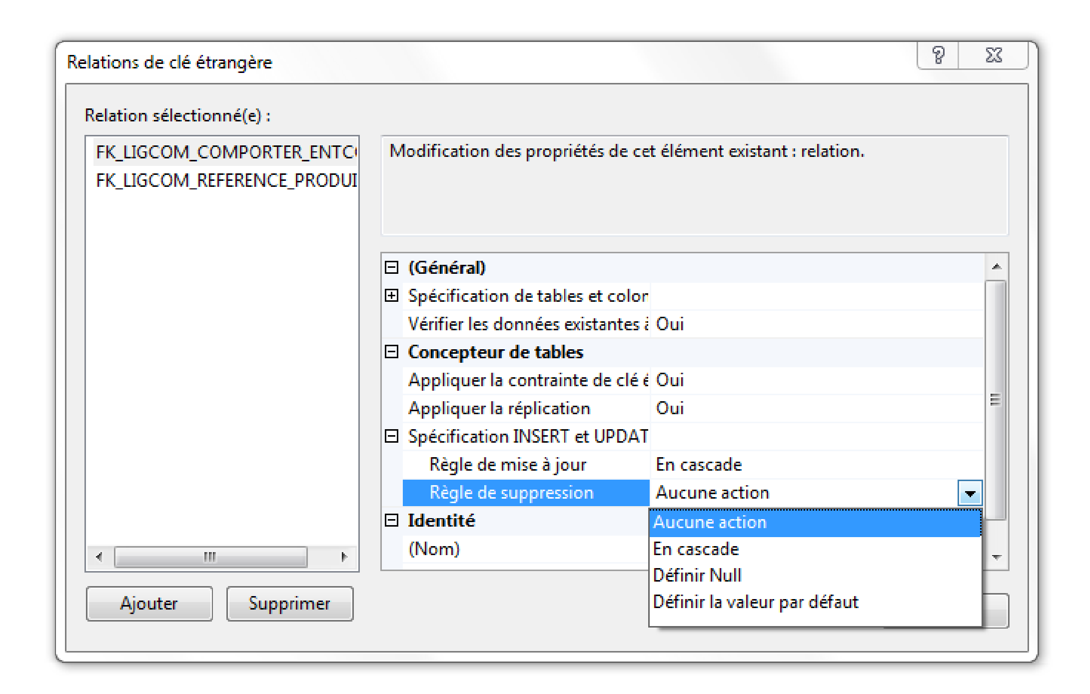
CODART	NUMFOU	DELLIV	QTE1	PRIX1	QTE2
B001	8700	15	0	150,0000	50
B002	8700	15	0	210,0000	50
D035	120	0	0	40,0000	0
D035	9120	5	0	40,0000	100
I105	120	90	10	705,0000	50
I105	540	70	0	810,0000	60
I105	8700	30	0	720,0000	50
I105	9120	60	0	920,0000	70
I105	9150	90	0	685,0000	90
I108	120	90	5	795,0000	30
I108	9120	60	0	920,0000	70
I10C	9120	60	0	800,0000	70
I10C	9150	90	0	650,0000	90
I10C	9180	30	0	720,0000	50
I110	9120	60	0	950,0000	70
I110	9180	90	0	900,0000	70
P220	120	15	0	3700,0000	100
P220	8700	20	50	3500,0000	100

C'est la même chose pour la table LICOM.

DELTA1.Papyrus - vente.LIGCOM | DELTA1.Papyrus - vente.VENDRE

NUMCOM	NUMLIG	CODART	QTELIV	DERLIV	QTECDE
1	1	I10C	3000	2010-03-15 00:...	3000
1	2	I105	2000	2011-07-05 00:...	2000
1	3	I108	1000	2011-08-20 00:...	1000
1	5	P220	6000	2011-02-20 00:...	200
1	6	P240	2000	2011-03-31 00:...	6000
2	1	I105	1000	2011-05-16 00:...	1000
1	4	D035	250	2011-02-20 00:...	200
3	1	B001	0	2011-12-31 00:...	200
3	2	B002	0	2011-12-31 00:...	200
4	1	I10C	1000	2011-05-15 00:...	1000
4	2	I105	500	2010-05-15 00:...	500
5	1	I10C	1000	2010-07-15 00:...	500
5	2	P220	10000	2010-08-31 00:...	10000
6	1	I110	50	2011-10-31 00:...	50
7	1	P230	12000	2010-11-01 00:...	15000
7	2	P220	1000	2010-11-10 00:...	1000
8	1	I105	200	2010-11-01 00:...	200

De la même manière, on pourrait appliquer les règles suivantes :



<u>Règle de suppression</u> : Spécifie ce qui se produit si un utilisateur tente de mettre à jour une ligne contenant des données impliquées dans une relation de clé étrangère.

- **Aucune action :** Un message d'erreur indique à l'utilisateur que la suppression n'est pas autorisée et la commande **DELETE** est annulée.
- **Cascade :** Supprime toutes les lignes contenant des données qui interviennent dans la relation de clé étrangère.
- **Définir Null** : définis la valeur null si toutes les colonnes clés étrangères de la table peuvent accepter des valeurs null.
- **Définir la valeur par défaut :** définis la valeur par défaut définie pour la colonne si toutes les colonnes clés étrangères de la table ont des valeurs par défaut définies.

<u>Règle de mise à jour</u> : Spécifie ce qui se produit si un utilisateur tente de mettre à jour une ligne contenant des données impliquées dans une relation de clé étrangère.

- **Aucune action :** Un message d'erreur indique à l'utilisateur que la mise à jour n'est pas autorisée et la commande **UPDATE** est annulée.
- **Cascade :** Mets à jour toutes les lignes contenant des données qui interviennent dans la relation de clé étrangère.
- **Définir Null :** définis la valeur null si toutes les colonnes clés étrangères de la table peuvent accepter des valeurs null.
- **Définir la valeur par défaut :** définis la valeur par défaut définie pour la colonne si toutes les colonnes clés étrangères de la table ont des valeurs par défaut définies.

Par le code

Nous pouvons modifier nos tables en utilisant les requêtes Tansact SQL. La modification de table est effectuée par la commande **ALTER TABLE**. Lors d'une modification de table, il est possible d'ajouter et de supprimer des colonnes et des contraintes, de modifier la définition d'une colonne (type de données, classement et comportement vis-à-vis de la valeur NULL), d'activer ou de désactiver les contraintes d'intégrité et les déclencheurs. Ce dernier point peut s'avérer utile lors d'import massif de données dans la base si l'on souhaite conserver des temps de traitements cohérents.

```
ALTER TABLE [nomSchema.] nomtable
{ [ ALTER COLUMN nom_colonne
{ nouveau_type_données [ ( longueur [ , precision ] ) ]
[ COLLATE classement ] [ NULL | NOT NULL ] } ]
| ADD nouvelle_colonne
| [ WITH CHECK | WITH NOCHECK ] ADD contrainte_table
| DROP { [ CONSTRAINT ] nom_contrainte | COLUMN nom_colonne }
| { CHECK | NOCHECK } CONSTRAINT { ALL | nom_contrainte }
| { ENABLE | DISABLE } TRIGGER { ALL | nom_déclencheur } }
```

nomSchema

Nom du schéma dans lequel la table va être définie.

WITH NOCHECK

Permet de poser une contrainte d'intégrité sur la table sans que cette contrainte soit vérifiée par les lignes déjà présentes dans la table.

COLLATE

Permet de définir un classement pour la colonne qui est différent de celui de la base de données.

NULL, NOT NULL

Permettent de définir une contrainte de nullité ou de non-nullité sur une colonne existante de la table.

CHECK, NOCHECK

Permettent d'activer et de désactiver des contraintes d'intégrité.

ENABLE, DISABLE

Permettent d'activer et de désactiver l'exécution des déclencheurs associés à la table.

L'ordre **ALTER** sur une table permet de :

• Supprimer une colonne
• Supprimer une contrainte
• Ajouter une colonne
• Ajouter une contrainte
• Ajouter une contrainte de ligne **DEFAULT**

Il ne permet pas de :

• Changer le nom d'une colonne
• Changer le type d'une colonne
• Ajouter une contrainte de ligne **NULL** / **NOT NULL**

Quelques exemples :

```
--Ajout d'une colonne
ALTER TABLE Clients ADD CODEREP char(2) null;
--Modification d'une colonne existante
ALTER TABLE Clients
      ALTER COLUMN Telephone char(14) not null;
--Ajouter une clé primaire sur la colonne COLONNE1
ALTER TABLE MATABLE1
ADD CONSTRAINT PK_PRIMARY
PRIMARY KEY (COLONNE1)
--Ajouter une clé étrangère à la table LIGCOM
ALTER TABLE LIGCOM
ADD CONSTRAINT FK_LIGCOM_COMPORTER_ENTCOM foreign key (NUMCOM)
      REFERENCES ENTCOM (NUMCOM)
GO
/* La contrainte FOREIGN KEY, basé sur la colonne NUMCOM référence le champ
NUMCOM de la table ENTCOM pour garantir que toutes lignes de commandes est
associé à un numéro de commande */
-- Ajout d'une contrainte
ALTER TABLE Clients
      ADD CONSTRAINT ck_cpo check(codepostal between 1000 and 95999);
```

Pour revenir à notre base de données « PAPYRUS ». Pour modifier le type de la colonne **DATCOM** de la table **ENTCOM**.

```
--Modification de la colonne DATCOM
ALTER TABLE ENTCOM
      ALTER COLUMN DATCOM smalldatetime not null;
```

On procédera de la même manière pour les différents types à modifier. On définira null ou not null pour la contrainte de nullité.

Définissons également les contraintes répondant à l'énoncé du problème : « Le numéro de commande est un champ compteur auto incrémenté de 1 ». Si vous souhaitez effectuer cette action par le code, vous devez supprimer puis recréer votre table en utilisant **IDENTITY** (1,1).

```
CREATE TABLE ENTCOM
(NUMCOM INT NOT NULL IDENTITY(1,1),
NUMFOU INT NOT NULL,
OBSCOM VARCHAR(25) NOT NULL,
DATCOM DATE NOT NULL,
CONSTRAINT PK_NUMCOM PRIMARY KEY (NUMCOM));
```

« La date de commande est par défaut la date du jour ». Pour la colonne **DATCOM** on utilisera la formule **getdate()**.

```
ALTER TABLE ENTCOM
      ADD CONSTRAINT DATCOM CHECK (DATCOM = getdate());
```

Le problème, c'est que nous avons contraint notre colonne à la date du jour. Il n'est par conséquent pas possible de saisir une autre date que la date du jour alors que si nous utilisons le code suivant :

```
ALTER TABLE ENTCOM
ADD CONSTRAINT DATCOM DEFAULT (getdate()) FOR DATCOM;
```

La colonne proposera la date du jour, mais celle-ci sera alors modifiable.

« L'indice de satisfaction est compris dans une échelle de 1 à 10 ».

```
ALTER TABLE ENTCOM
      ADD CONSTRAINT SATISFT CHECK (SATISF >= 1 AND SATISF <= 10);
```

« Le code postal est constitué de 5 chiffres ».

```
ALTER TABLE FOURNISSEUR
ADD CONSTRAINT POSFOU CHECK (POSFOU like ('[0-9][0-9][0-9][0-9][0-9]'));
```

Nous allons modifier les contraintes des tables LIGCOM, PRODUIT et VENDRE de façon à ce que lorsque nous modifierons le code d'un produit de la table PRODUIT (à l'aide d'une instruction **UPDATE**), la mise à jour se reporte automatiquement sur les tables LIGCOM et VENDRE. Pour cela nous spécifions la règle de mise à jour « **En cascade** ».

```
USE Papyrus
ALTER TABLE vente.LIGCOM
ADD CONSTRAINT FK_LIGCOM_UPDATE FOREIGN KEY (CODART)
REFERENCES vente.PRODUIT (CODART)
ON UPDATE CASCADE
```

```
USE Papyrus
ALTER TABLE vente.VENDRE
ADD CONSTRAINT FK_VENDRE_UPDATE FOREIGN KEY (CODART)
REFERENCES vente.PRODUIT (CODART)
ON UPDATE CASCADE
```

Une fois les relations de chacune des tables modifiées, nous allons mettre à jour notre table PRODUIT. Mettons à jour le code de l'article « I100 » par exemple.

	I100	Papier 1 ex continu	100	557	3500	B1000
	I105	Papier 2 ex continu	75	5	2300	B1000

```
Use Papyrus
UPDATE vente.PRODUIT
SET CODART = 'I10B'
WHERE CODART = 'I100'
```

Nous changeons le code de l'article en « I10B ». On enregistre les modifications et on actualise notre base de données.

▶	I10B	Papier 1 ex continu	100	557	3500	B1000
	I105	Papier 2 ex continu	75	5	2300	B1000

On interroge maintenant la table VENDRE. On constate que celle-ci a été modifiée automatiquement.

DELTA1.Papyrus - vente.VENDRE

CODART	NUMFOU	DELLIV	QTE1	PRIX1	QTE2
B001	8700	15	0	150,0000	50
B002	8700	15	0	210,0000	50
D035	120	0	0	40,0000	0
D035	9120	5	0	40,0000	100
I105	120	90	10	705,0000	50
I105	540	70	0	810,0000	60
I105	8700	30	0	720,0000	50
I105	9120	60	0	920,0000	70
I105	9150	90	0	685,0000	90
I108	120	90	5	795,0000	30
I108	9120	60	0	920,0000	70
I10B	9120	60	0	800,0000	70
I10B	9150	90	0	650,0000	90
I10B	9180	30	0	720,0000	50
I110	9120	60	0	950,0000	70
I110	9180	90	0	900,0000	70
P220	120	15	0	3700,0000	100
P220	8700	20	50	3500,0000	100
P230	120	30	0	5200,0000	100
P230	8700	60	0	5000,0000	50

C'est la même chose pour la table LICOM.

DELTA1.Papyrus - vente.LIGCOM

NUMCOM	NUMLIG	CODART	QTELIV	DERLIV	QTECDE
1	1	I10B	3000	2010-03-15 00:...	3000
1	2	I105	2000	2011-07-05 00:...	2000
1	3	I108	1000	2011-08-20 00:...	1000
1	5	P220	6000	2011-02-20 00:...	200
1	6	P240	2000	2011-03-31 00:...	6000
2	1	I105	1000	2011-05-16 00:...	1000
1	4	D035	250	2011-02-20 00:...	200
3	1	B001	0	2011-12-31 00:...	200
3	2	B002	0	2011-12-31 00:...	200
4	1	I10B	1000	2011-05-15 00:...	1000
4	2	I105	500	2010-05-15 00:...	500
5	1	I10B	1000	2010-07-15 00:...	500
5	2	P220	10000	2010-08-31 00:...	10000
6	1	I110	50	2011-10-31 00:...	50
7	1	P230	12000	2010-11-01 00:...	15000
7	2	P220	1000	2010-11-10 00:...	1000
8	1	I105	200	2010-11-01 00:...	200
NULL	NULL	NULL	NULL	NULL	NULL

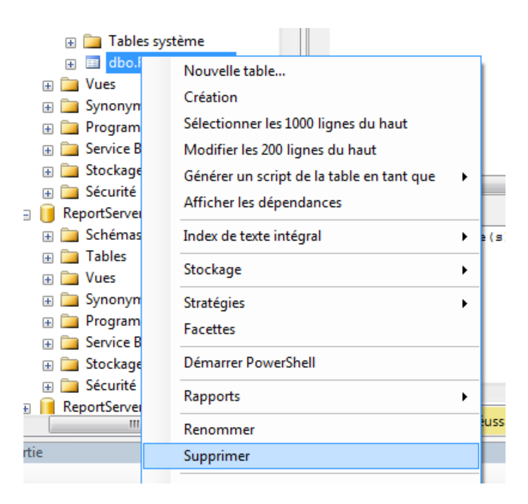
La contrainte **REFERENCE** traduit la liaison qui existe entre une clé primaire et étrangère de deux tables. Il est conseillé de créer ce genre de contrainte qu'après la création de toutes les tables impliquées, sinon, lors de la compilation de votre script, des erreurs peuvent apparaître. Cette contrainte n'a pas de propriété particulière par défaut, il faut les ajouter soi-même, voyons dans un premier temps sa syntaxe.

Supprimer une table

La suppression d'une table entraîne la suppression de toutes les données présentes dans la table. Les déclencheurs et les index associés à la table sont également supprimés. Il en est de même pour les permissions d'utilisation de la table. Par contre, les vues, procédures et fonctions qui référencent la table ne sont pas affectées par la suppression de la table. Si elles référencent la table supprimée, alors une erreur sera levée lors de la prochaine exécution.

Par l'interface

À partir de « Microsoft SQL Server Management Studio », cliquez droit sur la table que vous souhaitez supprimer, puis cliquez sur « **Supprimer** ».



Par le code

```
DROP TABLE [nomSchema.] nomtable [,nomtable...]
```

La suppression d'une table supprimera les données et les index associés. La suppression ne sera pas possible si la table est référencée par une clé étrangère.

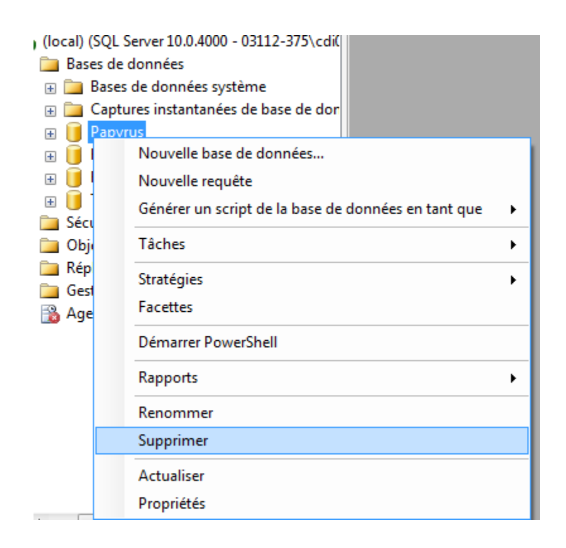
```
--Suppression de la table Client
DROP TABLE Client
```

Supprimer une base de données

La suppression d'une base de données entraîne la suppression de toutes vos données !

Par l'interface

À partir de « Microsoft SQL Server Management Studio », cliquez droit sur la base de données où vous souhaitez supprimer, puis cliquez sur « **Supprimer** ».



Par le code

Les bases de données sont détruites avec la commande **DROP DATABASE** :

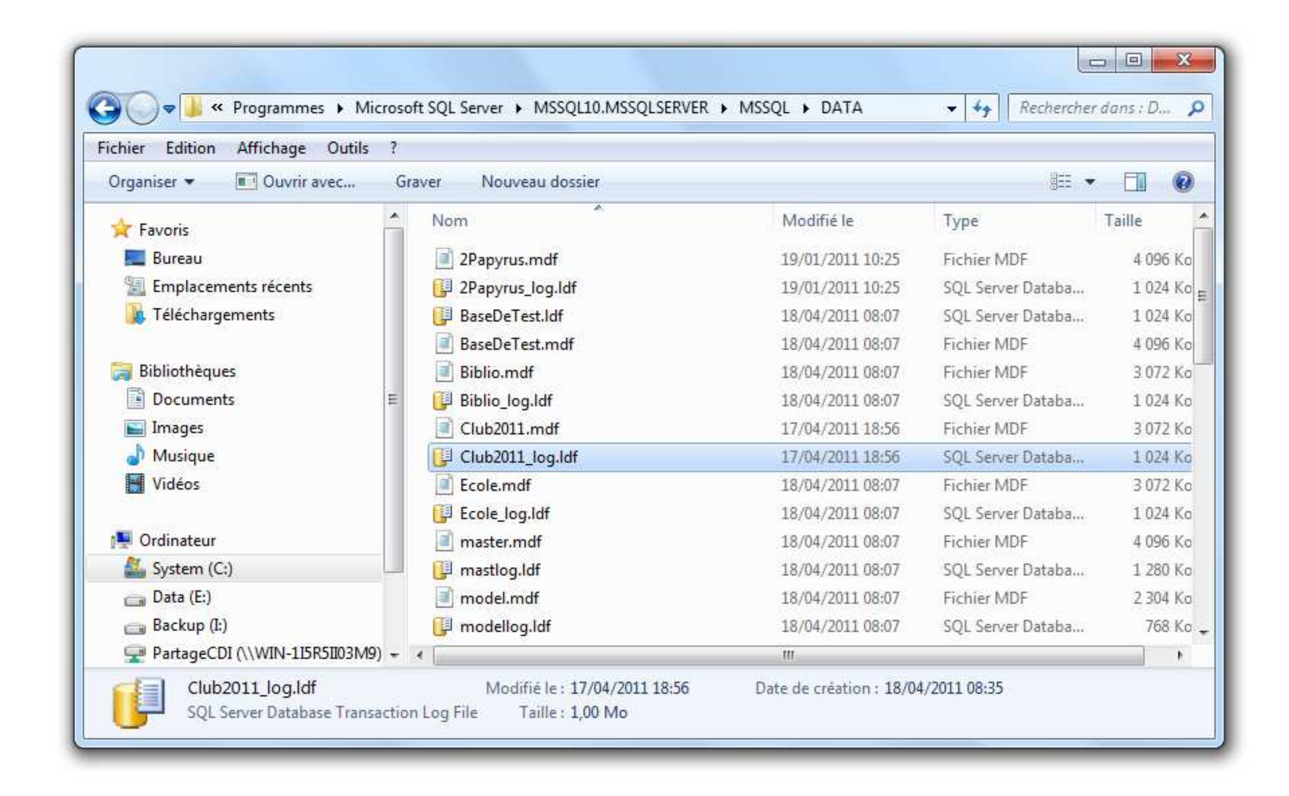
```
DROP DATABASE nom;
```

Seul le propriétaire de la base de données ou un super utilisateur peut supprimer une base de données. Supprimer une base de données supprime tous les objets qui étaient contenus dans la base. La destruction d'une base de données ne peut pas être annulée.

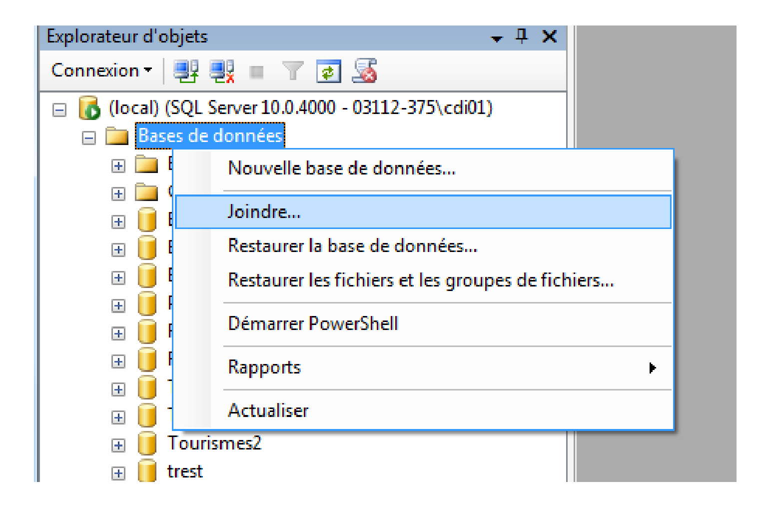
Vous ne pouvez pas exécuter la commande **DROP DATABASE** en étant connecté à la base de données cible. Néanmoins, vous pouvez être connecté à une autre base de données.

Joindre une base de données existante

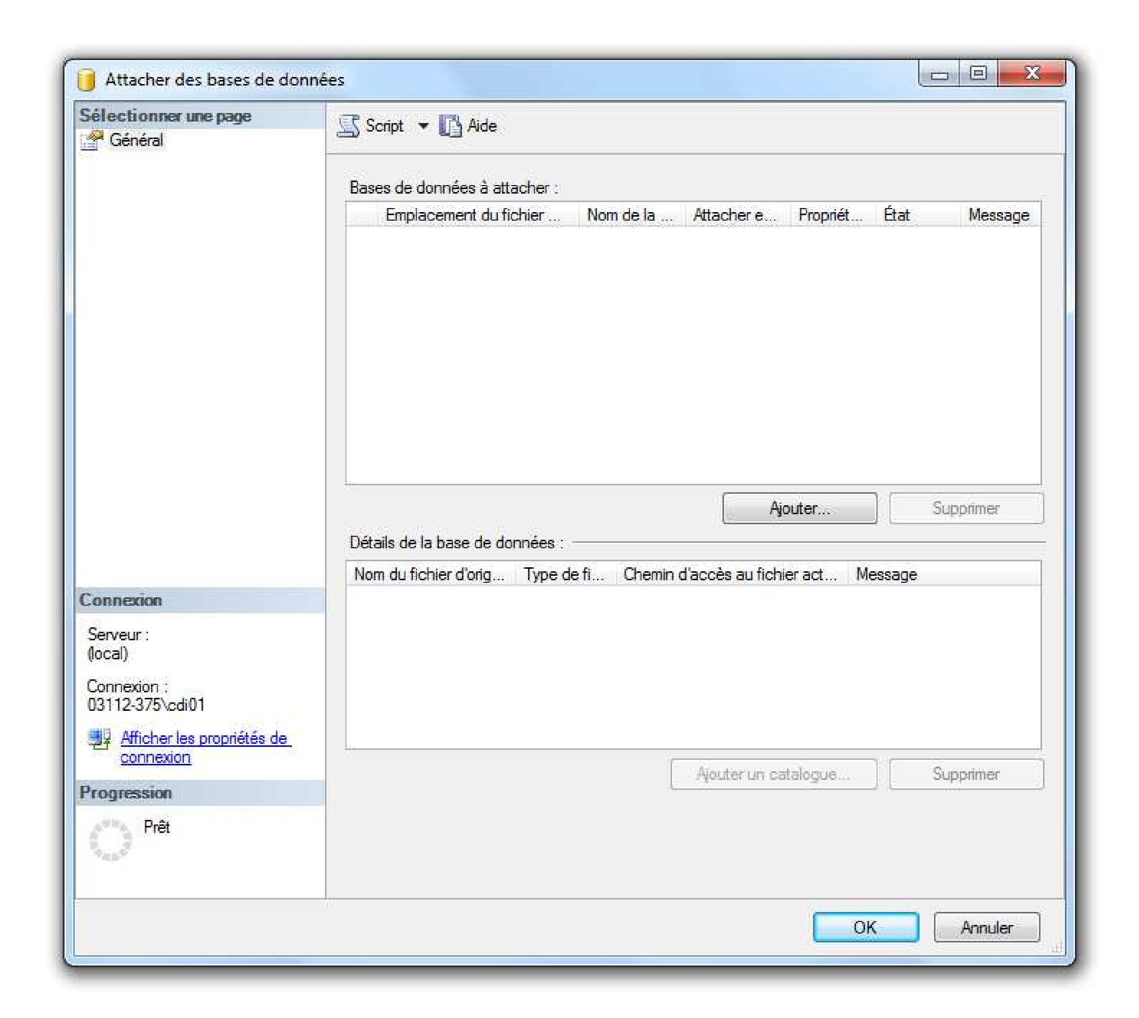
Vous pouvez joindre une base déjà existante à condition de posséder les fichiers **.mdf** et **.ldf** de la base de données originale. Placez ces deux fichiers dans le répertoire « **C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA** » (selon la version de votre Microsoft SQL Server). Dans notre exemple, nous avons copié dans le répertoire, les fichiers « **Club2011.mdf** » et « **Club2011_log.ldf** ».



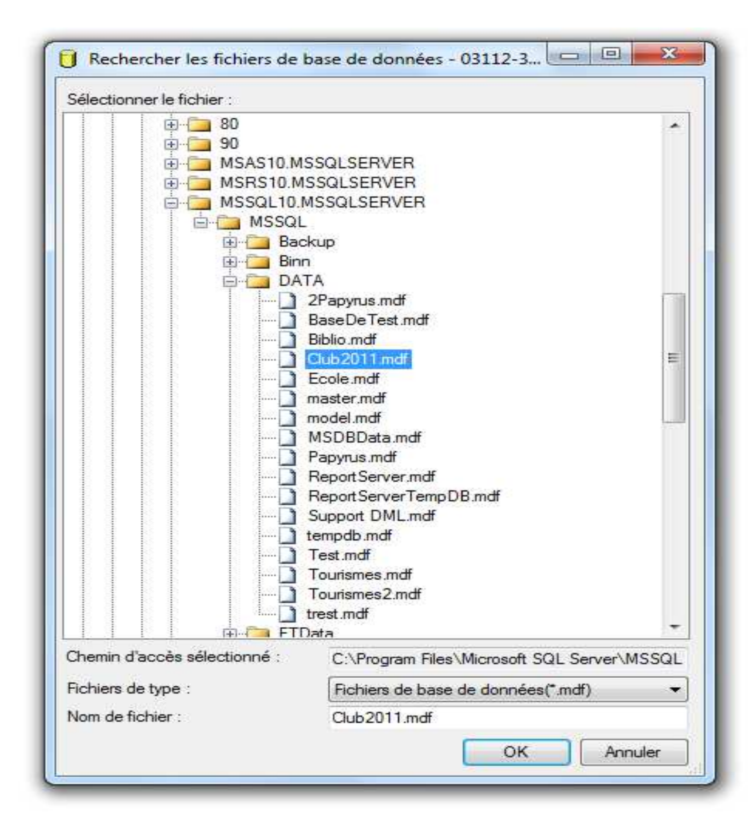
Sous « **Microsoft SQL Server** », cliquez droit sur le dossier « **Bases de données** » puis « **Joindre…** ».



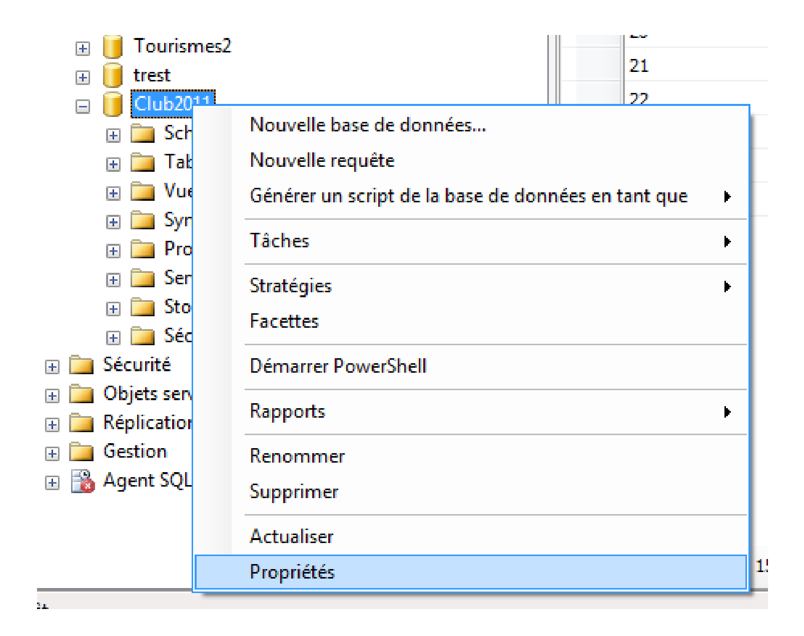
Cliquez sur « **Ajouter…** ».



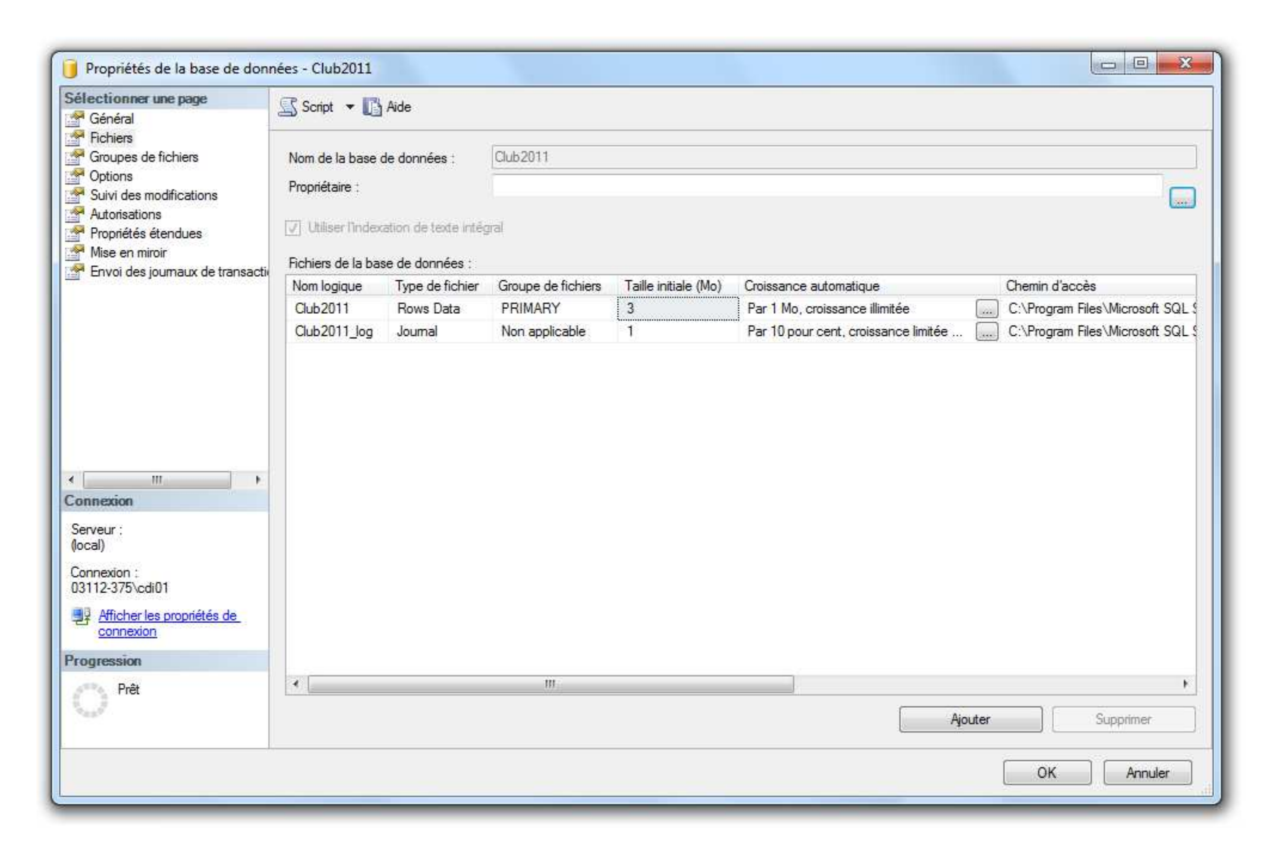
Et rechercher le fichier « **Club2011.mdf** » (pour notre exemple).



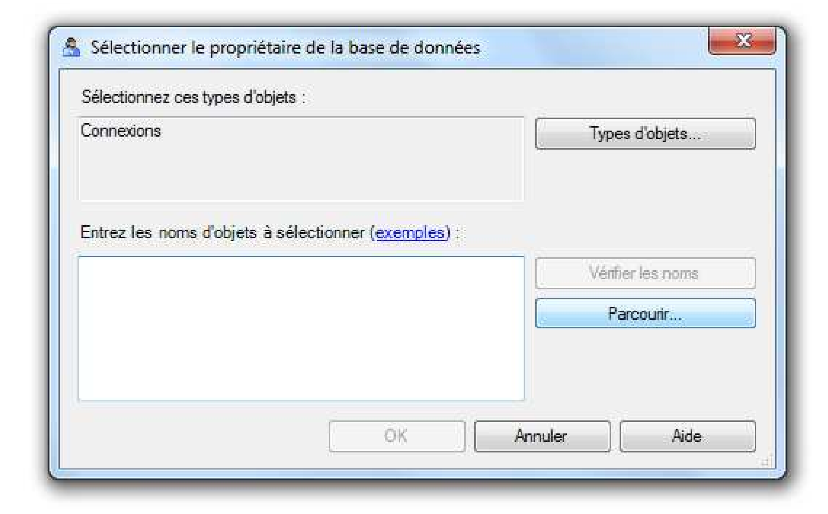
La base de données est maintenant rattachée. Vous devez vous rendre propriétaire de cette base. Pour cela, cliquez droit sur la base concernée, puis « **Propriétés** ».



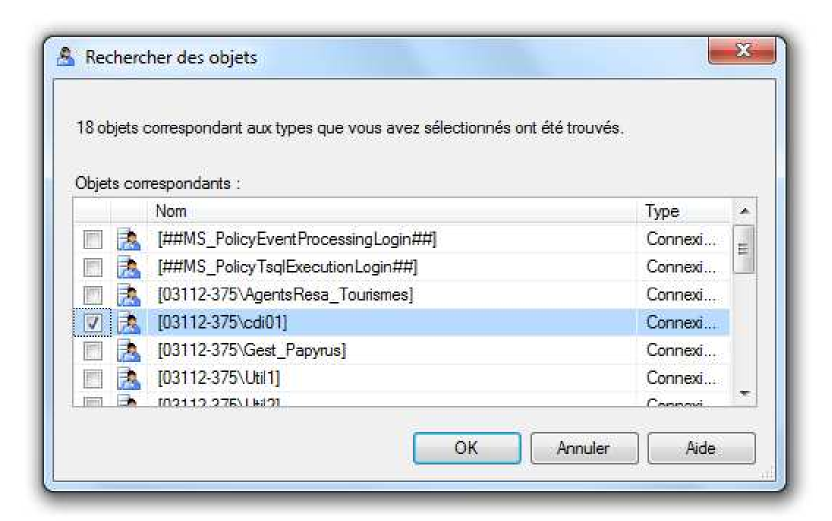
Cliquez sur « **...** » au niveau du propriétaire.



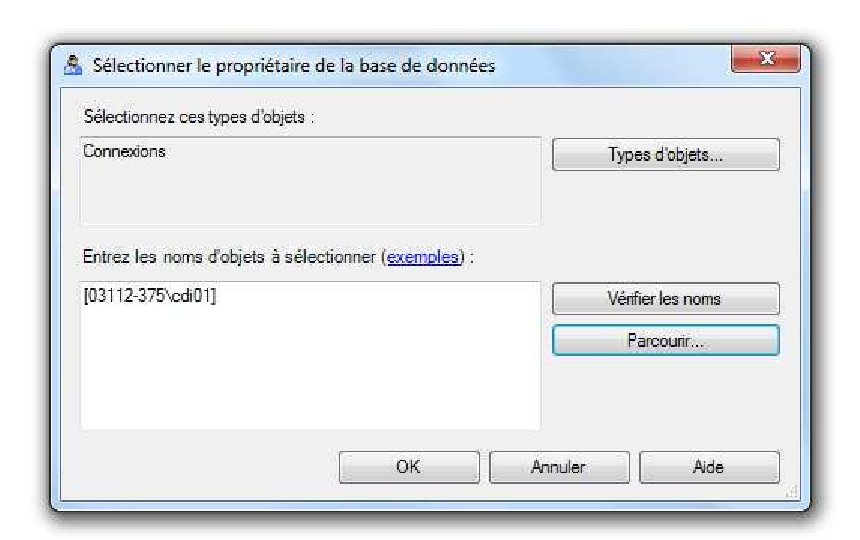
Puis « **Parcourir** ».



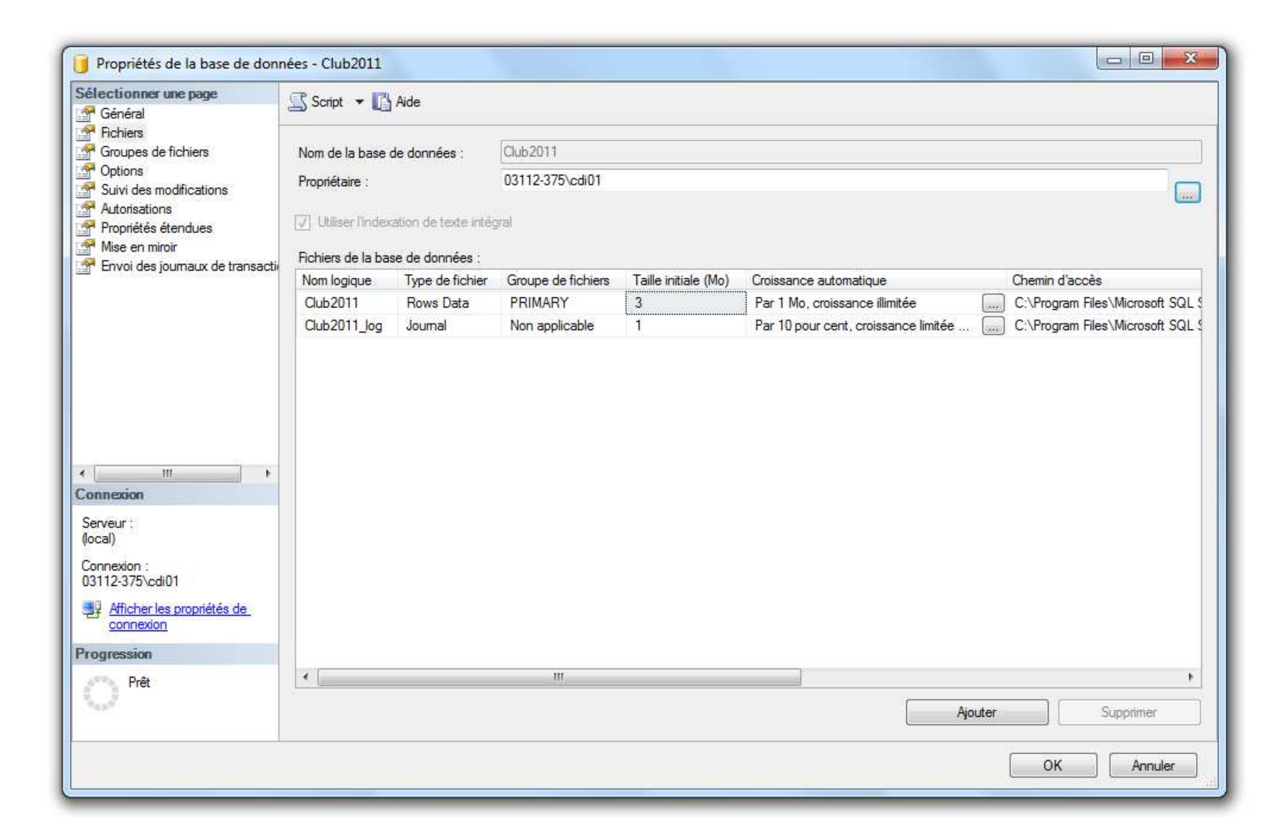
Vous devriez vous retrouver dans la liste. Cochez la case vous correspondant.



Cliquez sur « **Ok** » pour terminer.



Vous apparaissez maintenant comme propriétaire de la base.



Groupes de fichiers

Les groupes de fichiers sont essentiellement destinés aux bases de données importantes (> 1 GO) et aux tâches d'administrations avancées : le principal intérêt est d'améliorer le temps de réponse de la base, en autorisant la création de ses fichiers sur plusieurs disques et / ou l'accès par de multiples contrôleurs de disques.

Les objets et les fichiers de la base de données peuvent être regroupés en groupes de fichiers à des fins d'allocation de pages et d'administration. Il existe deux types de groupes de fichiers.

Primaires

Le groupe de fichiers primaire contient le fichier de données primaire ainsi que tous les autres fichiers qui ne sont pas spécifiquement affectés à un autre groupe de fichiers. Toutes les pages des tables système sont allouées au groupe de fichiers primaire.

Définis par l'utilisateur

Ce groupe désigne tous les groupes de fichiers créés à l'aide du mot clé **FILEGROUP** dans une instruction **CREATE DATABASE** ou **ALTER DATABASE**.

Les fichiers journaux ne font jamais partie d'un groupe de fichiers. L'espace qui leur est réservé est géré indépendamment de l'espace réservé aux données.

Un fichier ne peut pas appartenir à plusieurs groupes de fichiers. Les tables, les index et les données LOB (Large Object) peuvent être associés à un groupe de fichiers particulier. Dans ce cas, toutes leurs pages sont allouées à ce groupe de fichiers ou les tables et les index peuvent être partitionnés. Les données des index et des tables partitionnées sont réparties en unités, dont chacune peut être placée dans un groupe de fichiers distinct au sein d'une base de données.

Dans chaque base de données, un groupe de fichiers est désigné comme groupe de fichiers par défaut. Lorsque vous créez une table ou un index sans spécifier un groupe de fichiers, le système considère que toutes les pages doivent être allouées depuis le groupe de fichiers par défaut. Seul un groupe de fichiers à la fois peut faire office de groupe de fichiers par défaut. Les membres du rôle de base de données fixe **db_owner** peuvent faire basculer le groupe de fichiers par défaut d'un groupe à un autre. En l'absence de spécification, c'est le groupe de fichiers primaire qui sert de groupe de fichiers par défaut.

<u>Exemple</u> : On crée une base de données sur une instance de SQL Server.

```
USE master;
GO
CREATE DATABASE BaseDeTest
ON PRIMARY
  ( NAME='BaseDeTest_dat',
    FILENAME=
       'c:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\data\BaseDeTest.mdf',
    SIZE=4MB,
    MAXSIZE=10MB,
    FILEGROWTH=1MB)
LOG ON
  ( NAME='BaseDeTest_log',
    FILENAME =
       'c:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\data\BaseDeTest.ldf',
    SIZE=1MB,
    MAXSIZE=10MB,
    FILEGROWTH=1MB);
GO
```

Nous allons créer 5 groupes de fichiers. On crée les espaces de stockage du partitionnement :

```
-- Création des storages :
ALTER DATABASE BaseDeTest
   ADD FILEGROUP Groupe1;

ALTER DATABASE BaseDeTest
   ADD FILEGROUP Groupe2;

ALTER DATABASE BaseDeTest
   ADD FILEGROUP Groupe3;

ALTER DATABASE BaseDeTest
   ADD FILEGROUP Groupe4;
```

```
ALTER DATABASE BaseDeTest
   ADD FILEGROUP Groupe5;

-- Ajouts de fichiers aux storages :
ALTER DATABASE BaseDeTest
   ADD FILE (NAME        = 'Partitionnement1',
             FILENAME    = 'c:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\data\Partitionnement1.ndf',
             SIZE        = 1 GB,
             FILEGROWTH = 10 MB)
TO FILEGROUP Groupe1;

ALTER DATABASE BaseDeTest
   ADD FILE (NAME        = 'Partitionnement2',
             FILENAME    = 'c:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\data\Partitionnement2.ndf',
             SIZE        = 1 GB,
             FILEGROWTH = 10 MB)
TO FILEGROUP Groupe2;

ALTER DATABASE BaseDeTest
   ADD FILE (NAME        = 'Partitionnement3',
             FILENAME    = 'c:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\data\Partitionnement3.ndf',
             SIZE        = 1 GB,
             FILEGROWTH = 10 MB)
TO FILEGROUP Groupe3;

ALTER DATABASE BaseDeTest
   ADD FILE (NAME        = 'Partitionnement4',
             FILENAME    = 'c:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\data\Partitionnement4.ndf',
             SIZE        = 1 GB,
             FILEGROWTH = 10 MB)
TO FILEGROUP Groupe4;

ALTER DATABASE BaseDeTest
   ADD FILE (NAME        = 'Partitionnement5',
             FILENAME    = 'c:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\data\Partitionnement5.ndf',
             SIZE        = 1 GB,
             FILEGROWTH = 10 MB)
TO FILEGROUP Groupe5;
```

5 storages ont été créés et dans chacun de ces espaces de stockage on a créé un fichier de 1 Go avec une stratégie de croissance par pas de 10 Mo. Notez que chaque fichier est créé sur le même disque. Dans le cas où nous créons des fichiers sur des disques différents, l'accès doit être physique comme une partition de disque par exemple.

Les partitions

Le partitionnement de table introduit depuis Microsoft SQL Server 2005 permet aux utilisateurs de découper de grandes tables en plusieurs structures de stockage. Par défaut, les tables ne comportent qu'une seule partition.

Le fait de diviser, physiquement, les données d'une table, d'une vue ou d'un index constitue le partitionnement. La division se fait horizontalement (par lignes) et permet ainsi de diviser les grosses tables en petits morceaux mieux gérables.

Pourquoi partitionner ?

Imaginez que vous ayez une table de commandes qui reçoit chaque jour des milliers de lignes, au fil des années, cette table commencera à devenir très grosse et très peu gérable (dans le cas où vous voudriez ajouter un index, ou modifier une colonne…). En fait, la table deviendra en définitive un gros bloc monolithique intouchable...

Pourquoi ne pas avoir la possibilité de diviser cette table entre plusieurs disques durs ? Ainsi, nous pourrions mettre les commandes du mois dans un disque dur à faible capacité, mais à très grande vitesse et les données "archives" dans un énorme disque de plusieurs centaines de Go. Pourquoi ne pas mettre une grosse table dans plein de petits disques durs ? Nous pourrions ainsi bénéficier de parallélisme dans la lecture et l'écriture des données. C'est pour ces raisons que le partitionnement existe !

Le partitionnement se fait en trois phases :

Le processus de partitionnement d'une table, d'un index ou d'une vue indexée se déroule en trois étapes :

. Créer une fonction de partition
. Créer un schéma de partition mappé sur une fonction de partition
. Créer la table, l'index, ou la vue indexée sur le schéma de partition

Fonction de partition

Une fonction de partitionnement est totalement indépendante de toute structure physique ou logique de la table de données. Son rôle est de définir des "points de partitionnement" ou les valeurs qui délimitent les partitions.

Une fonction de partition permet de spécifier la manière dont la table est partitionnée à l'aide de l'instruction Transact-SQL **CREATE PARTITION FUNCTION**.

Pour répartir les données entre les différentes partitions, la fonction utilise des plages de valeur. Seules les valeurs frontières sont indiquées.

```
CREATE PARTITION FUNCTION nom_fonction ( parametre )
AS RANGE [ LEFT | RIGHT ]
FOR VALUES ( [ valeurlimite [ ,...n ] ] )
```

parametre

Spécifie le type de colonne : Tous, sauf **timestamp, varchar(max), nvarchar(max)** et **varbinary**

LEFT | RIGHT

LEFT indique que la valeur frontière est incluse dans la plage de valeurs inférieures, **RIGHT** dans la plage suivante.

valeurlimite

Marque la frontière de chaque partition

Quelques exemples :

```
CREATE PARTITION FUNCTION
maPremiereFonction (INT)
AS RANGE RIGHT FOR VALUES (1000, 2000, 3000);
```

Cette fonction à comme paramètre un entier et dit : « Je partitionne dès que l'entier que tu m'as spécifié atteint 1000, 2000 ou 3000 ». La clause **AS RANGE LEFT|RIGHT** montre juste à quelle partition va aller le point de partitionnement.

Dans l'exemple suivant, nous créons 3 valeurs pivots portant sur des dates.

```
CREATE PARTITION FUNCTION DateCommande (DATE)
AS
RANGE RIGHT
FOR VALUES ('2009-01-01', '2010-01-01', '2011-01-01');
```

On a donc créé 3 valeurs pivot ce qui impose au moins 4 partitions :

- Partition 1, de - l'infini à 2008-12-31
- Partition 2, de 2009-01-01 à 2009-12-31
- Partition 3, de 2010-01-01 à 2010-12-31
- Partition 4, de 2011-01-01 à + l'infini

RANGE RIGHT signifie que la valeur borne est incluse dans la partition à droite.

Schéma de partitionnement

Un schéma de partitionnement spécifie les zones de partitionnement physiques (alors que la fonction de partitionnement désigne les points logiques) et les associe à la fonction de partitionnement.

On crée un schéma de partition pour faire correspondre les groupes de fichiers qui contiendront les partitions aux valeurs indiquées par la fonction de partition à l'aide de l'instruction Transact-SQL **CREATE PARTITION SCHEME**.

```
CREATE PARTITION SCHEME nom_schema_partition
AS PARTITION nom_fonction_partition
[ ALL ] TO ( { file_group_name | [ PRIMARY ] } [ ,...n ] )
```

Un exemple : Utilisation de la fonction de partition définie précédemment pour subdiviser la table en 3 partitions basées sur 3 groupes de fichiers déjà existants dans la base.

```sql
CREATE PARTITION SCHEME partSchema
AS PARTITION maPremiereFonction
TO (Groupe1, Groupe2, Groupe3)
```

Cette commande effectue les opérations suivantes :

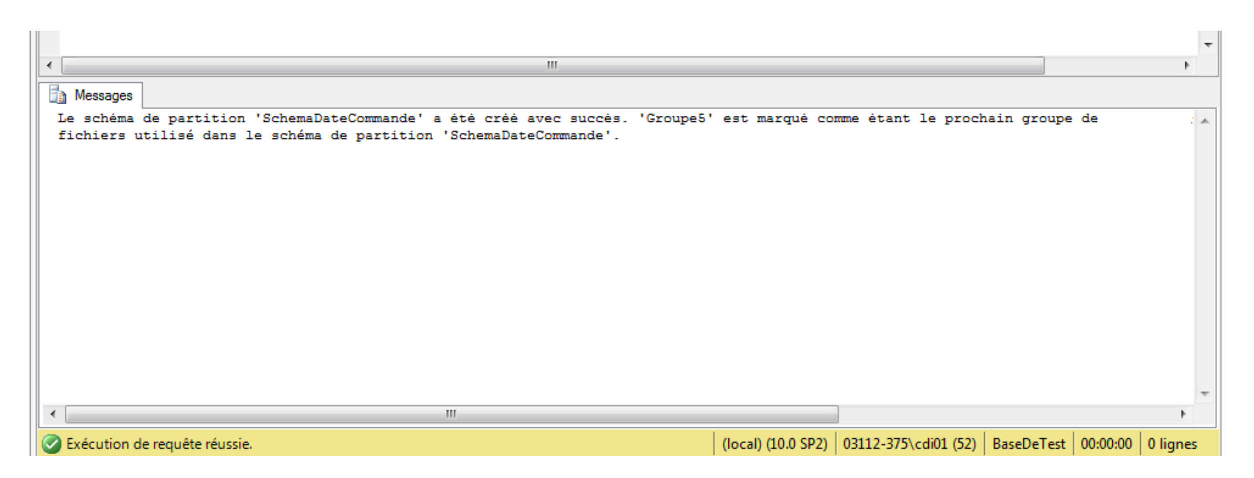
Les données entre 0 et 1000 seront stockées dans Groupe1,
Les données entre 1001 et 2000 seront stockées dans Groupe2,
Les données entre 2001 et l'infini seront stockés dans Groupe3.

Si la proposition "les données entre x et y" n'est pas clair, c'est normal, c'est la troisième étape.

SQL Server 2008 permet l'utilisation du mot clé **ALL** qui autorise toutes les partitions définies par la fonction de partition à être créées dans un seul groupe de fichiers. Si vous n'utilisez pas le mot clé **ALL**, alors le schéma de partition doit contenir au moins un groupe de fichiers pour chaque partition définie dans la fonction de partition.

Prenons l'exemple de notre base de données « **BaseDeTest** » où nous avions créé 5 groupes. Nous allons créer un schéma de partitionnement en utilisant la fonction DateCommande vu à la section précèdent.

```sql
CREATE PARTITION SCHEME SchemaDateCommande
AS PARTITION DateCommande
TO (Groupe1, Groupe2, Groupe3, Groupe4, Groupe5);
```



Notez que le schéma de partitionnement fait référence à la fonction de partitionnement. Il inclut les 5 espaces de stockage définis précédemment, c'est-à-dire les 4 nécessaires au schéma de partitionnement, et un storage supplémentaire de réserve pour des travaux ultérieurs.

Partitionner des tables et des index

Dernière et ultime étape, nous allons comprendre pourquoi nous avons fait les étapes 1 et 2. La syntaxe pour créer une table partitionnée est la suivante :

```sql
CREATE TABLE
```

```
[ database_name . [ schema_name ] . | schema_name . ] table_name
( { <column_definition> | <computed_column_definition> }
[ <table_constraint> ] [ ,...n ] )
[ ON { partition_scheme_name ( partition_column_name ) | filegroup
| "default" } ]
[ { TEXTIMAGE_ON { filegroup | "default" } ] [ ; ]
```

Un exemple :

```
CREATE TABLE MaTable (
PK INT IDENTITY(1,1) PRIMARY KEY,
NOM NVARCHAR(50),
PRENOM NVARCHAR(80)
)
ON partSchema(PK)
GO
```

Ce que nous venons de faire : Créer une table toute bête et lui associer le schéma de partitionnement crée précédemment avec la clé primaire (le int) comme paramètre de fonction de partitionnement. En clair : Tous les enregistrements avec clé primaire inférieure ou égale à 1000 iront dans le Groupe1, les enregistrements avec 1001 inférieur à PK et inférieur ou égale à 2000 dans Groupe2…

Reprenons notre base de données « **BaseDeTest** ». Nous avons créé une fonction de partition, un schéma de partition mappé sur la fonction de partition. Dernière étape :

```
CREATE TABLE Commandes
(NUMCOM SMALLINT NOT NULL IDENTITY(1,1),
NOMCLIENT VARCHAR(30) NOT NULL,
PRENOMCLIENT VARCHAR (35) NOT NULL,
DATCOM DATE NOT NULL)
ON SchemaDateCommande(DATCOM)
```

La table créée possède une colonne qui servira de partitionnement ayant un type **DATE** et non N**ULLable**.

C'est cette colonne qui est passée en argument de la fonction de partition de la clause ON pour préciser que la table est stockée sur un système partitionné.

Imaginez juste ça à plus grande échelle avec des groupes dans des disques séparés. Voilà l'utilité du partitionnement !

SELECT sur des tables partitionnées

Vous avez la possibilité de faire des requêtes sur des partitions ! Vu que le partitionnement est un mécanisme tout à fait transparent pour les personnes utilisant les tables, il serait embêtant pour vous, DBA qui avez partitionné la table, de parcourir toute la table avec un **SELECT** en sachant pertinemment que les données que vous cherchez sont dans une partition X.

Il est possible de cibler une partition (ou plusieurs) dans votre **SELECT** grâce à l'instruction **$PARTITION**. Exemple :

```
SELECT * FROM MaTable WHERE $PARTITION.ma_fonction_de_partition(cle) = 3
```

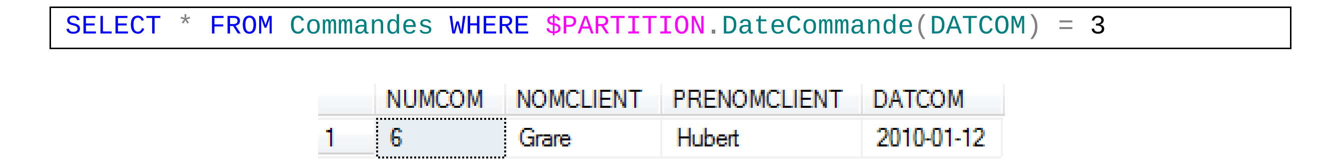
Cette instruction nous retourne tous les enregistrements de la partition numéro 3.

Prenons un exemple concret, celui de la base de données « **BaseDeTest** ». Nous nous baserons sur le jeu d'essai suivant :

NUMCOM	NOMCLIENT	PRENOMCLIENT	DATCOM
4	Grare	Stéphane	2008-01-09
1	Leblanc	Gerard	2009-11-10
3	Tardieu	Michael	2011-11-10
5	Grare	Fabien	2007-10-09
6	Grare	Hubert	2010-01-12

Nous souhaitons connaître toutes les lignes de la table Commandes de la partition 3.

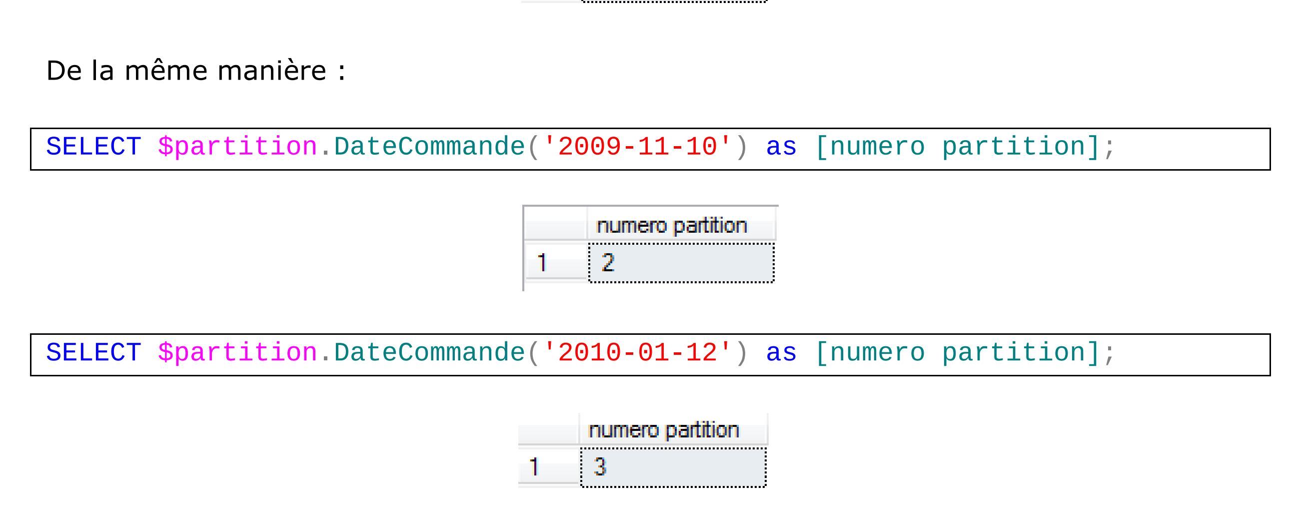
```
SELECT * FROM Commandes WHERE $PARTITION.DateCommande(DATCOM) = 3
```



	NUMCOM	NOMCLIENT	PRENOMCLIENT	DATCOM
1	6	Grare	Hubert	2010-01-12

Nous voulons savoir sur quel numéro de partition est stocké l'enregistrement concernant la date de commande du 09/01/2008 :

```
SELECT $partition.DateCommande('2008-01-09') as [numero partition];
```



	numero partition
1	1

De la même manière :

```
SELECT $partition.DateCommande('2009-11-10') as [numero partition];
```

	numero partition
1	2

```
SELECT $partition.DateCommande('2010-01-12') as [numero partition];
```

	numero partition
1	3

```
SELECT $partition.DateCommande('2011-11-10') as [numero partition];
```

	numero partition
1	4

On constate donc que l'enregistrement des commandes respecte donc bien notre partitionnement qui est pour rappel :

- Partition 1, de - l'infini à 2008-12-31
- Partition 2, de 2009-01-01 à 2009-12-31
- Partition 3, de 2010-01-01 à 2010-12-31
- Partition 4, de 2011-01-01 à + l'infini

Mais comment connaître le numéro de la partition ??? Et bien c'est tout simple, la requête suivante vous donne tous les numéros de partition de votre table :

```
SELECT $partition.DateCommande(DATCOM) AS [numeroPart], count(*) [nombre de
lignes]
FROM Commandes GROUP BY $partition.DateCommande(DATCOM)
```

	numeroPart	nombre de lignes
1	1	2
2	2	1
3	3	1
4	4	1

Gestion du partitionnement

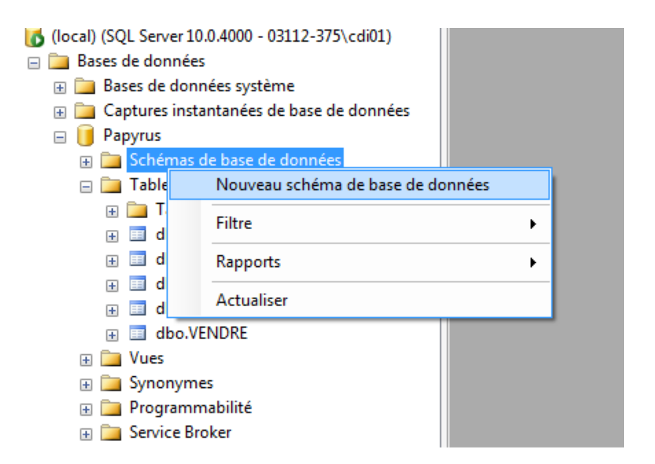
Un système de partitionnement n'a pas d'intérêt sans les outils pour l'administrer. Pour ce faire, il doit être possible de :

- Rajouter une partition (**ALTER PARTITION FUNCTION** ... **SPLIT RANGE** ...)
- Fusionner deux partitions contiguës (**ALTER PARTITION FUNCTION** ... **MERGE RANGE** ...)
- Voire, échanger le contenu de deux partitions (**ALTER TABLE** ... **SWITCH PARTITION** ...)

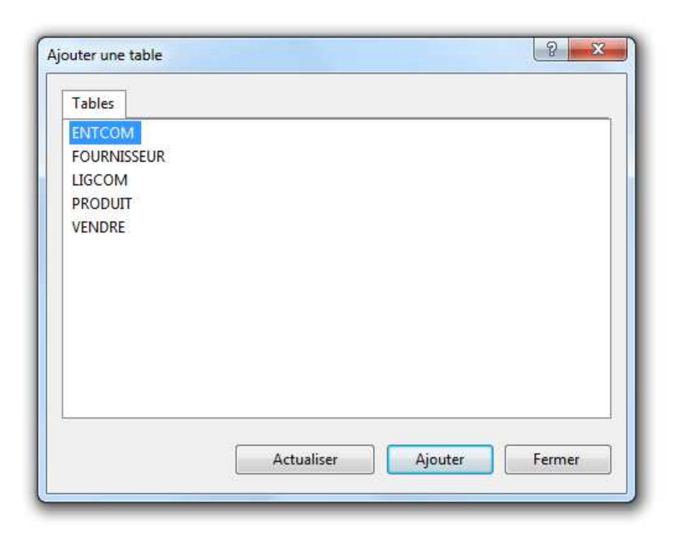
Sous MS SQL Server, ces opérations étant faites à bas niveau, SQL Server rend la main immédiatement après le lancement de ces ordres SQL. Les utilisateurs peuvent continuer à travailler comme si de rien n'était.

Schémas de la base de données

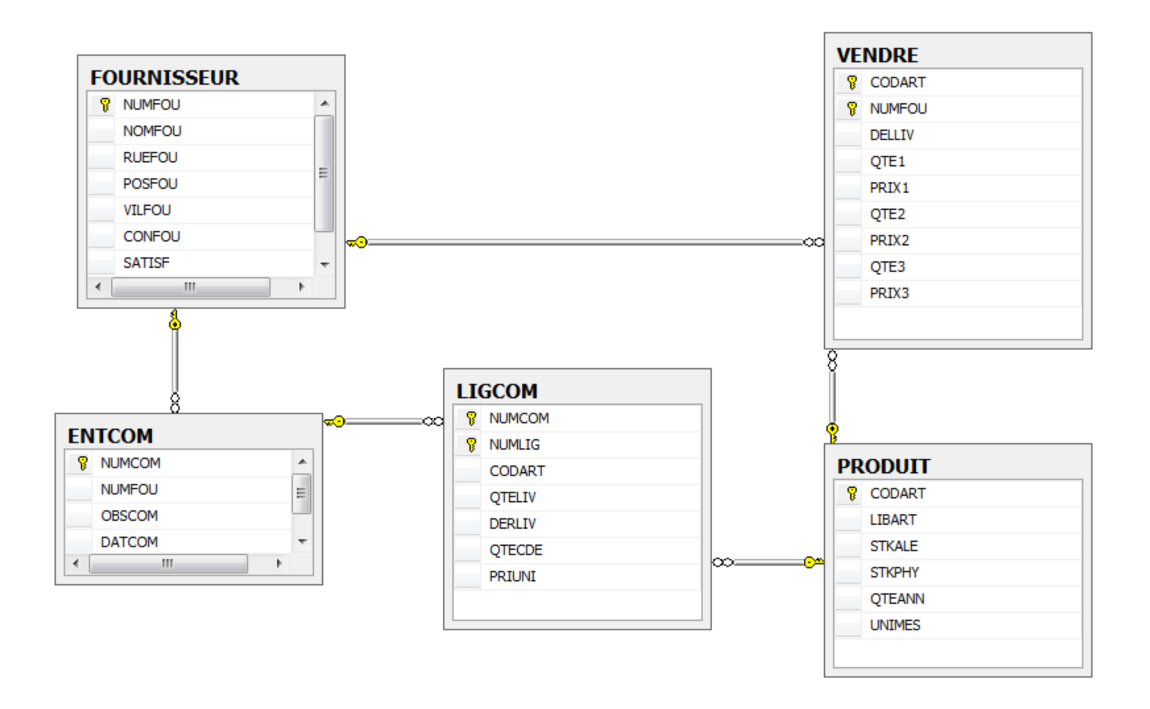
Pour obtenir le schéma de la base de données, cliquez droit sur « **Schémas de base de données** » de votre base de données « **Papyrus** » puis cliquez sur « **Nouveau schéma de base de données** ».



Ajoutez toutes les tables sur votre schéma.



Vous devez alors obtenir le schéma suivant :



Saisir des données dans vos tables

Pour saisir des données dans vos tables, nous allons vous présenter 3 méthodes. Nous utiliserons le jeu d'essai suivant :

➔ **La table Produit**

Code	***Libellé***	***Stock*** alerte	***Stock*** en-cours	***Qté*** annuelle	***Unité*** mes.
I100	Papier 1 ex continu	100	557	3500	B1000
I105	Papier 2 ex continu	75	5	2300	B1000
I108	Papier 3 ex continu	200	557	3500	B500
I110	Papier 4 ex continu	10	12	63	B400
P220	Pré imprimé commande	500	2500	24500	B500
P230	Pré imprimé facture	500	250	12500	B500
P240	Pré imprimé bulletin paie	500	3000	6250	B500
P250	Pré imprimé bon livraison	500	2500	24500	B500
P270	Pré imprimé bon fabrication	500	2500	24500	B500
R080	Ruban Epson 850	10	2	120	unité
R132	Ruban imp1200 lignes	25	200	182	unité
B002	Bande magnétique 6250	20	12	410	unité
B001	Bande magnétique 1200	20	87	240	unité
D035	CD R slim 80 mm	40	42	150	B010
D050	CD R-W 80mm	50	4	0	B010

➔ **La table ENTCOM**

Numéro commande	***Observation commande***	***Date commande***	***N° compte*** fournisseur
00010		10/02/2010	00120
00011	Commande urgente	01/03/2010	00540
00020		25/04/2010	09180
00025	Commande urgente	30/04/2010	09150
00210	Commande cadencée	05/05/2010	00120
00300		06/06/2010	09120
00250	Commande cadencée	02/10/2010	08700
00620		02/10/2010	00540
00625		09/10/2010	00120
00629		12/10/2010	09180

➔ La table LIGCOM

N° commande	N° Lig	Produit	Quantité cdée	Prix Unitaire	Qté livrée	Dernière Livraison
00010	01	I100	3000	47.00	3000	15/03/2010
00010	02	I105	2000	48.50	2000	05/07/2010
00010	03	I108	1000	68.00	1000	20/08/2010
00010	04	D035	200	40.00	250	20/02/2010
00010	05	P220	6000	350.00	6000	31/03/2010
00010	06	P240	6000	200.00	2000	31/03/2010
00011	01	I105	1000	60.00	1000	16/05/2010
00020	01	B001	200	140.00		31/12/2010
00020	02	B002	200	140.00		31/12/2010
00025	01	I100	1000	59.00	1000	15/05/2010
00025	02	I105	500	59.00	500	15/05/2010
00210	01	I100	1000	47.00	1000	15/07/2010
00010	02	P220	10000	350.00	10000	31/08/2010
00300	01	I110	50	79.00	50	31/10/2010
00250	01	P230	15000	490.00	12000	15/12/2010
00250	02	P220	10000	350.00	10000	10/11/2010
00620	01	I105	200	60.00	200	01/11/2010
00625	01	I100	1000	47.00	1000	15/10/2010
00625	02	P220	10000	350.00	10000	31/10/2010
00629	01	B001	200	140.00		31/12/2010
00629	02	B002	200	140.00		31/12/2010

➔ La table Fournisseur

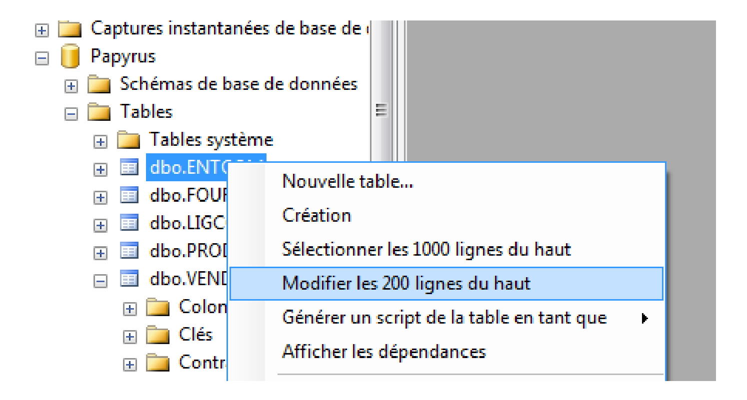
N° compte fournisseur	Raison sociale	Adresse	Nom Contact	indice satisfaction
00120	GROBRIGAN	20 rue du papier 92200 papercity	Georges	08
00540	ECLIPSE	53, rue laisse flotter les rubans 78250 Bugbugville	Nestor	07
08700	MEDICIS	120 rue des plantes 75014 Paris	Lison	
09120	DISCOBOL	11 rue des sports 85100 La Roche sur Yon	Hercule	08
09150	DEPANPAP	26, avenue des locomotives 59987 Coroncountry	Pollux	05
09180	HURRYTAPE	68, boulevard des octets 04044 Dumpville	Track	

➔ La table Vente

Code	N° cpt fourn.	Délai livr.	Qté 1	Prix 1	Qté 2	Prix 2	Qté 3	Prix 3
I100	00120	90	0	700	50	600	120	500
I100	00540	70	0	710	60	630	100	600
I100	09120	60	0	800	70	600	90	500
I100	09150	90	0	650	90	600	200	590
I100	09180	30	0	720	50	670	100	490
I105	00120	90	10	705	50	630	120	500
I105	00540	70	0	810	60	645	100	600
I105	09120	60	0	920	70	800	90	700
I105	09150	90	0	685	90	600	200	590
I105	08700	30	0	720	50	670	100	510
I108	00120	90	5	795	30	720	100	680
I108	09120	60	0	920	70	820	100	780
I110	09180	90	0	900	70	870	90	835
I110	09120	60	0	950	70	850	90	790
D035	00120	0	0	40				
D035	09120	5	0	40	100	30		
I105	09120	8	0	37				
D035	00120	0	0	40				
D035	09120	5	0	40	100	30	5	0
I105	09120	8	0	37				
P220	00120	15	0	3700	100	3500		
P230	00120	30	0	5200	100	5000		
P240	00120	15	0	2200	100	2000		
P250	00120	30	0	1500	100	1400	500	1200
P250	09120	30	0	1500	100	1400	500	1200
P220	08700	20	50	3500	100	3350		
P230	08700	60	0	5000	50	4900		
R080	09120	10	0	120	100	100		
R132	09120	5	0	275				
B001	08700	15	0	150	50	145	100	140
B002	08700	15	0	210	50	200	100	185

Par l'interface

Cliquez droit sur la table dans laquelle vous souhaitez ajouter des données puis « **Modifier les 200 lignes du haut**… ».



Vous pouvez alors compléter vos données ligne par ligne.

03112-375.Papyrus - dbo.ENTCOM

	NUMCOM	NUMFOU	OBSCOM	DATCOM
	8	120	Null	2011-01-18 11:...
	9	540	Commande urge...	2011-01-18 11:...
	10	9180	NULL	2011-01-18 11:...
	11	9150	Commande urge...	2011-01-18 11:...
▶	12	120	ommande cadencé	2011-01-18 11:...
*	NULL	NULL	NULL	NULL

De la même manière, vous pouvez modifier et supprimer les lignes de votre table.

Par le code

Nous allons vous présenter les instructions principales : **INSERT** pour l'insertion de donnée, **UPDATE** pour la mise à jour de vos données et **DELETE** pour la suppression. Nous verrons également d'autres commandes possibles.

→ INSERT

La création de lignes dans une table, ou dans une vue selon certaines conditions se fait par la commande **INSERT**.

Exemple d'insertion par ligne de code :

```
INSERT INTO PRODUIT (CODART, LIBART, STKALE, STKPHY, QTEANN, UNIMES)
VALUES ('P250', 'Pré imprimé bon livraison', '500', '2500', '24500',
'B500')
```

```
INSERT INTO LIGCOM (NUMCOM, NUMLIG, CODART, QTELIV, DERLIV, QTECDE, PRIUNI)
VALUES ('00010' , '04', 'I100', '250', '20/02/2010', '200', '40')
```

Ici on ajoute plusieurs lignes en même temps :

```
INSERT INTO LIGCOM (NUMCOM, NUMLIG, CODART, QTELIV, DERLIV, QTECDE, PRIUNI)
VALUES ('00010' , '04', 'I100', '250', '20/02/2010', '200', '40'),
      ('00010' , '05', 'I100', '350', '20/02/2010', '100', '50'),
      ('00010' , '04', 'I100', '150', '20/02/2010', '400', '40')
```

Autres exemples :

Insérer l'employé 00140, de nom REEVES, de prénom HUBERT dans le département A00, de salaire 2100€.

```
INSERT INTO EMPLOYES
VALUES (00140,'REEVES','HUBERT','A00', 2100)
```

Insérer dans la table EMPLOYES_A00 préalablement créée de structure tous les employés de la table EMPLOYES attachés au département A00.

```
INSERT INTO EMPLOYES_A00 (NOEMP, NOM, PRENOM, SALAIRE)
SELECT NOEMP, NOM, PRENOM, SALAIRE
```

```
FROM EMPLOYES
WHERE WDEPT = 'A00'
```

➔ UPDATE

La modification des valeurs des colonnes de lignes existantes s'effectue par l'instruction **UPDATE**. Cette instruction peut mettre à jour plusieurs colonnes de plusieurs lignes d'une table à partir d'expressions ou à partir de valeurs d'autres tables.

Augmenter le salaire de 20% de tous les employés pour la table EMPLOYES de structure

```
UPDATE EMPLOYES
SET SALAIRE = SALAIRE * 1,2
```

Augmenter le salaire de 20% de l'employé de matricule 00040.

```
UPDATE EMPLOYES
SET SALAIRE = SALAIRE * 1,2
WHERE NOEMP = 00040
```

SET

Nom des colonnes et leurs valeurs ou expressions mises à jour.

FROM

Nom d'autres tables utilisées pour fournir des critères.

WHERE

Critère de sélection pour la mise à jour d'une ligne (optionnel) Si la clause **WHERE** n'est pas codée, la table entière sera mise à jour.

Augmenter le salaire de 20% des employés du service informatique (repéré par son nom dans la table DEPART), pour les tables EMPLOYES et DEPART.

```
UPDATE EMPLOYES
SET SALAIRE = SALAIRE * 1,2
WHERE DEPT IN (SELECT NODEPT FROM DEPART
WHERE NOMDEPT LIKE 'SERVICE%')
```

➔ DELETE

La suppression des valeurs des colonnes existantes s'effectue par l'instruction **DELETE**.

Supprimer tous les employés de la table EMPLOYES.

```
DELETE FROM EMPLOYES
```

Supprimer les employés du département 'E21'

```
DELETE FROM EMPLOYES
WHERE WDEPT ='E21'
```

FROM

Spécifie le nom de la table ou les lignes seront supprimées.

FROM

Une deuxième clause **FROM** pour spécifier le nom d'autres tables utilisées pour fournir des critères.

WHERE

Spécifie le critère de sélection (optionnel) si la clause **WHERE** n'est pas codée, toutes les lignes seront supprimées.

Supprimer tous les employés du service informatique (repéré par son nom dans la table DEPART), pour les tables EMPLOYES et DEPART.

```
WHERE DEPT IN (SELECT NODEPT FROM DEPART
WHERE NOMDEPT LIKE 'SERVICE%')
```

Supprimer tous les employés du service informatique pour les tables EMPLOYES et DEPART.

```
DELETE FROM EMPLOYES
FROM EMPLOYES
JOIN DEPART
ON DEPT = NODEPT
WHERE NOMDEPT LIKE 'SERVICE%'
```

➔ **TRUNCATE TABLE**

Pour supprimer rapidement toutes les lignes d'une table, **TRUNCATE TABLE** est plus efficace que l'utilisation d'une instruction **DELETE** sans clause **WHERE** : Plus rapide et utilisation de moins de ressources du système et du journal des transactions.

```
TRUNCATE TABLE <nom_table>
```

TRUNCATE TABLE ne peut être utilisée que pour une suppression de toutes les lignes de la table, et sur des tables non référencées par une **Foreign Key**.

Supprimer tous les employés de la table EMPLOYES

```
TRUNCATE TABLE EMPLOYES
```

➔ **MERGE**

MERGE est une nouveauté de SQL Server 2008, qui permet d'effectuer plusieurs opérations **INSERT**, **UPDATE** et **DELETE** en une seule instruction. En fonctions de conditions, l'instruction MERGE peut :

. Mettre à jour une ligne si elle existe, ou insérer les données dans une nouvelle ligne

. Synchroniser le contenu de 2 tables

Soit la table RECAP qui est alimentée toutes les nuits par un processus lisant les commandes du jour : Si le produit commandé existe dans la table RECAP, la quantité totale est incrémentée de la quantité commandée, s'il n'existe pas, une nouvelle ligne est créée dans la table RECAP.

```
MERGE RECAP as Cible
USING (SELECT IDProduit, QteCommandee FROM DETAILCOMMANDE
WHERE DateCde = CAST(GETDATE() as DATE)) as Source
ON ( Cible.IDProduit = Source.IDProduit)
WHEN MATCHED
THEN UPDATE Set Cible.QteTotale = Cible.QteTotale + Source.QteCommandee
WHEN NOT MATCHED
THEN INSERT (IDProduit, QteTotale)
VALUES(Source.IDProduit,Source.QteCommandee);
--Une instruction MERGE doit se terminer par un point-virgule
```

La clause **INTO** spécifie la table ou vue qui est la cible des opérations d'insertion, de mise à jour ou de suppression.

La clause **USING…ON** spécifie la table ou la requête à appliquer en tant que conditions de recherche pour les critères correspondants.

Les clauses **WHEN** spécifient les actions à prendre en fonction des résultats de la clause **USING…ON**.

- o **WHEN MATCHED** indique que la condition est vérifiée
- o **WHEN NOT MATCHED [BY TARGET]** indique que la condition n'est pas vérifiée du fait de la cible
- o **WHEN NOT MATCHED [BY SOURCE]** indique que la condition n'est pas vérifiée du fait de la source

La clause **OUTPUT** retourne une ligne pour chaque ligne dans l'objet cible mis à jour, inséré ou supprimé.

➔ OUTPUT

SQL Server crée et gère automatiquement les tables inserted et deleted qui ont la même structure que la table sur laquelle porte la requête **INSERT**, **UPDATE**, **DELETE** ou **MERGE**.

Avant SQL Server 2005, on pouvait accéder à ces tables uniquement à partir d'un déclencheur. On peut désormais accéder à ces tables directement dans le cadre d'une instruction **INSERT**, **UPDATE**, **DELETE** ou **MERGE**, grâce à la clause **OUTPUT**.

La clause **OUTPUT** permet de travailler avec ces tables de 2 manières :

- Elle permet de retourner leur contenu directement à l'application, en tant qu'ensemble de résultats.
- Elle permet d'insérer leur contenu dans une table ou variable de table.

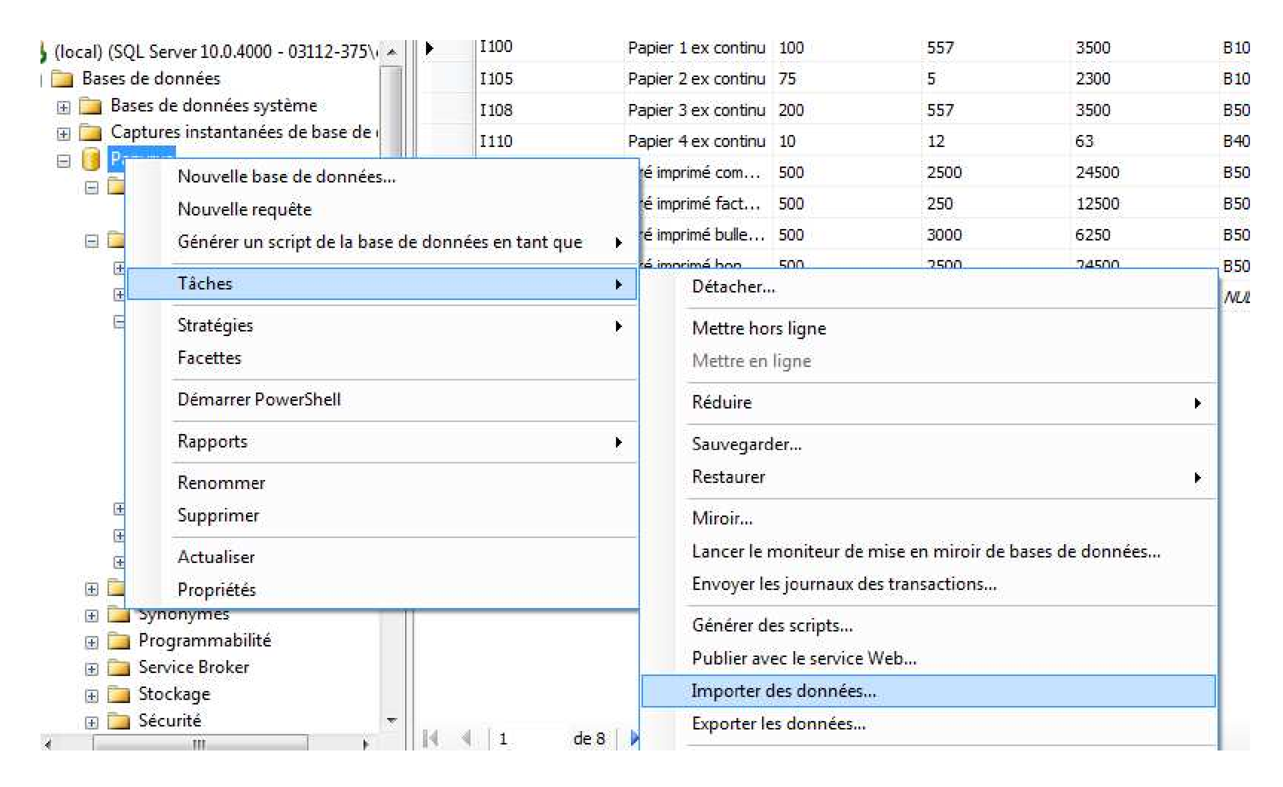
Par l'option insertion de SQL Server

Vous avez la possibilité d'importer des données à partir d'un fichier Excel. Vous devez veiller à retrouver toutes les colonnes correspondantes à vos différentes tables, de préférence, dans l'ordre. Attention, les données présentes dans le fichier Excel ne doivent pas être en doublon avec les données contenues dans vos tables. Dans notre exemple, nous créons un fichier Excel avec deux onglets. Chaque onglet représente une table. Ici nous avons la table PRODUIT et la table VENDRE.

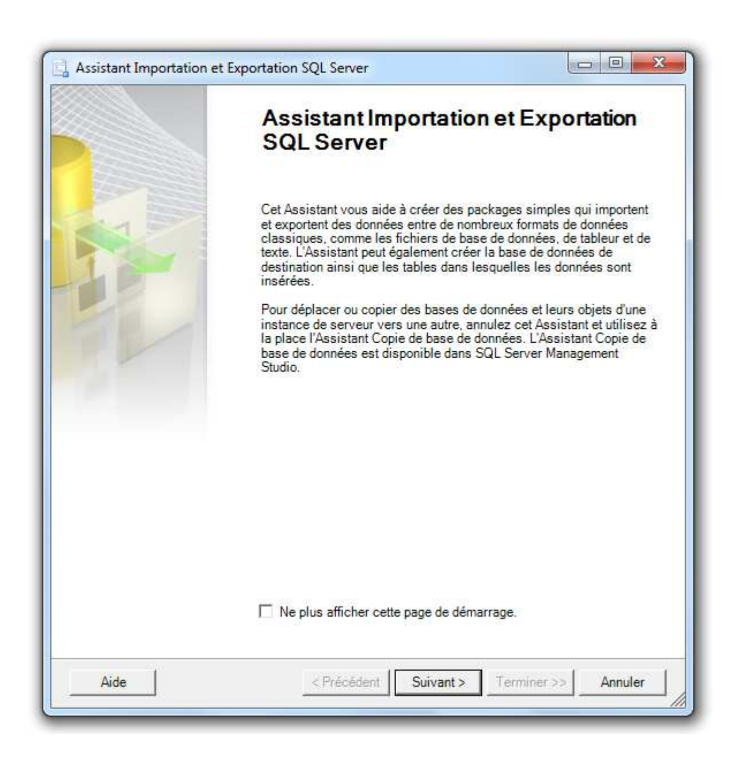
	A	B	C	D	E	F
1	CODART	LIBART	STKALE	STKPHY	QTEANN	UNIMES
2	B001	Bande magn	20	87	240	unité
3	B002	Bande magn	20	12	410	unite
4	D035	CD R slim 80	40	42	150	B010
5	D050	CD R-W 80m	50	4	0	B010
6	P270	Pré-imprimé	500	2500	24500	B500
7	R080	ruban Epson	10	2	120	unite
8	R132	ruban impl 12	25	200	182	unite
9						

Produit | Vente

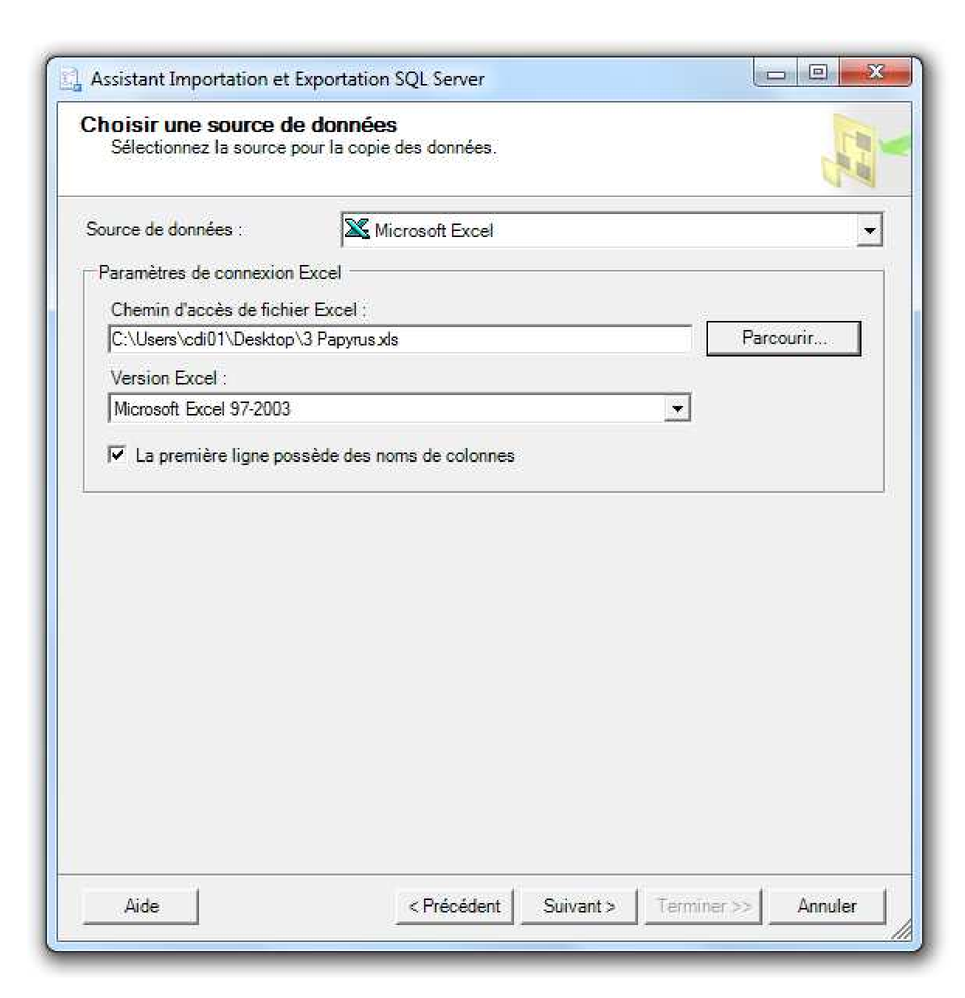
À partir de « Microsoft SQL Server Management Studio », cliquez droit sur la base de données dans laquelle vous souhaitez importer des données, puis cliquer sur « **Tâches** » et « **Importer des données…** ».



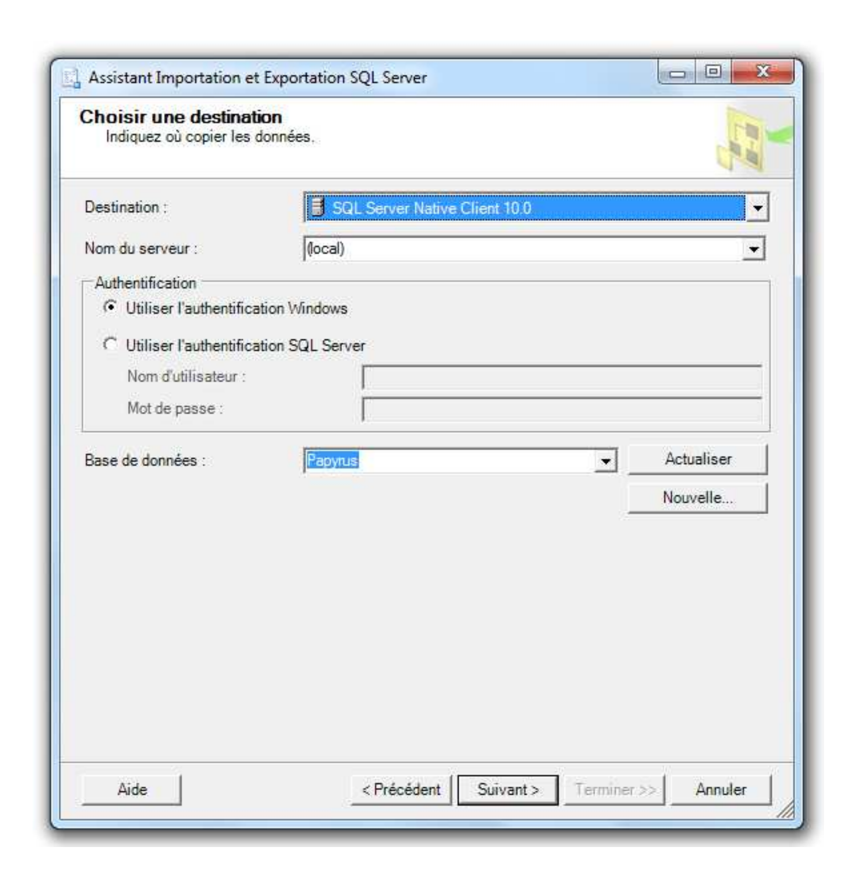
L'assistant importation et exportation de SQL Server s'ouvre alors. Cliquez alors sur suivant.



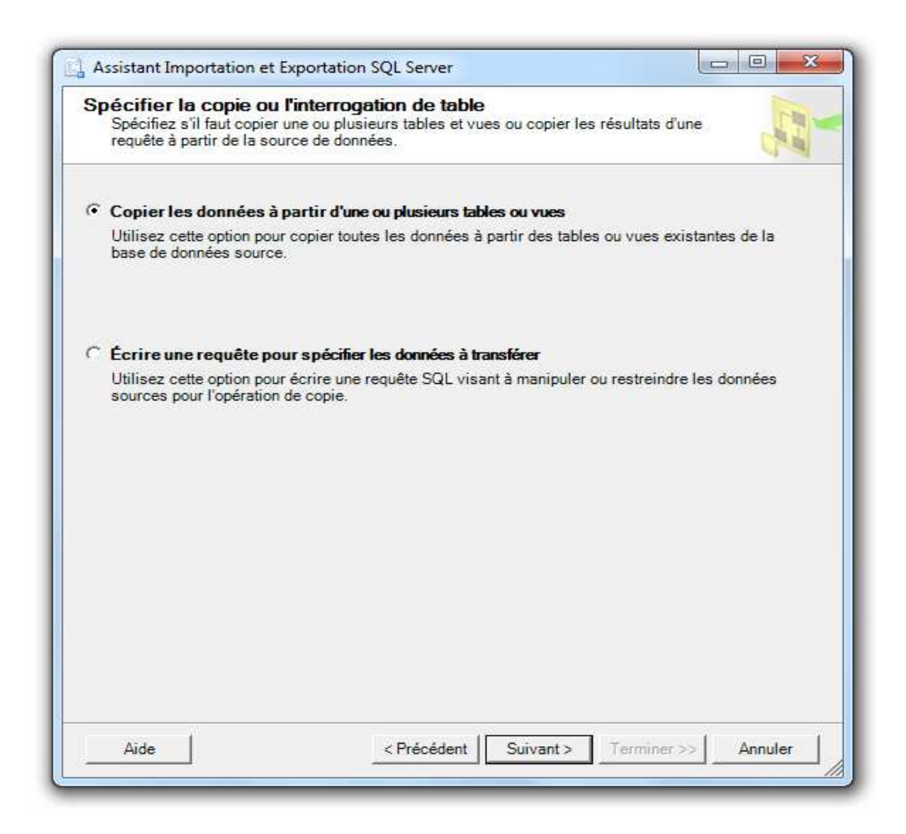
Sélectionnez la source de données (on indiquera qu'il s'agit d'une source Microsoft Excel) puis indiquez-lui le chemin où se trouve votre fichier.



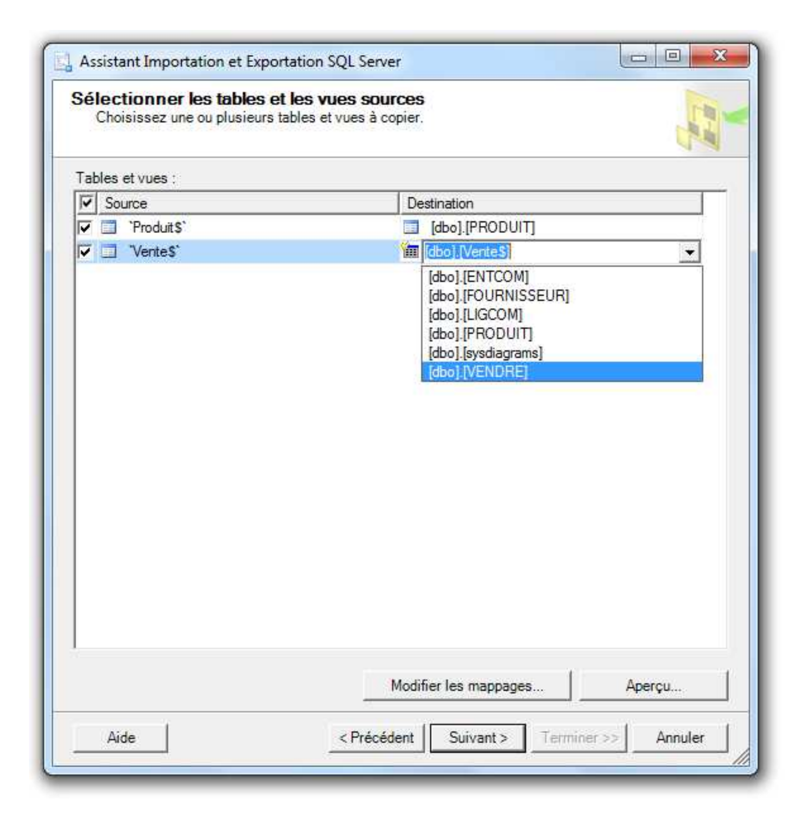
Choisissez une destination. Dans notre exemple, on ne modifie rien. On clique sur suivant.



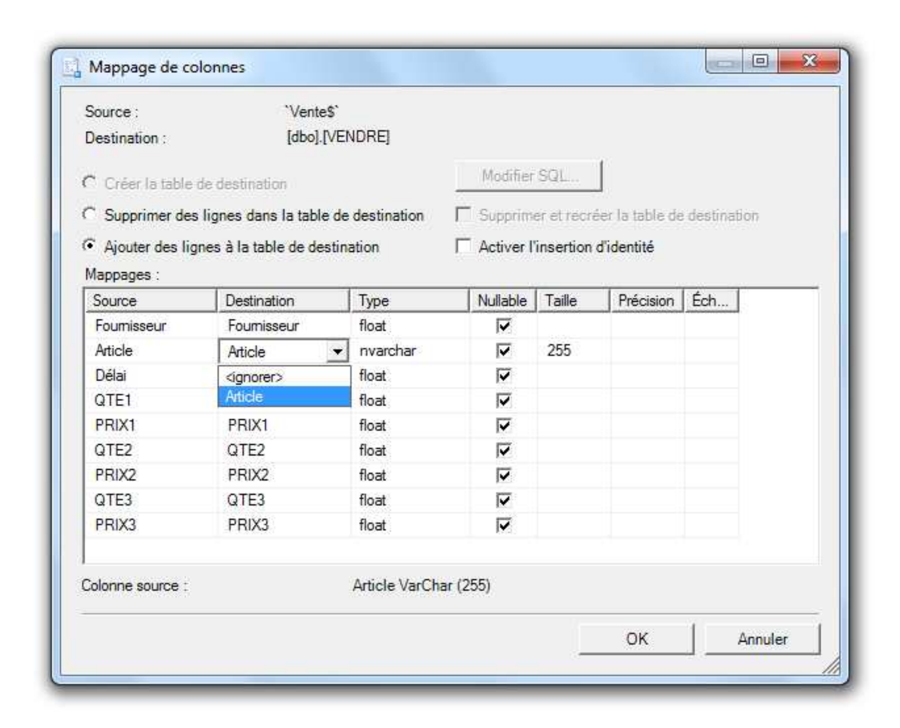
On sélectionne « **Copier les données à partir d'une ou plusieurs tables ou vue** » puis suivant.



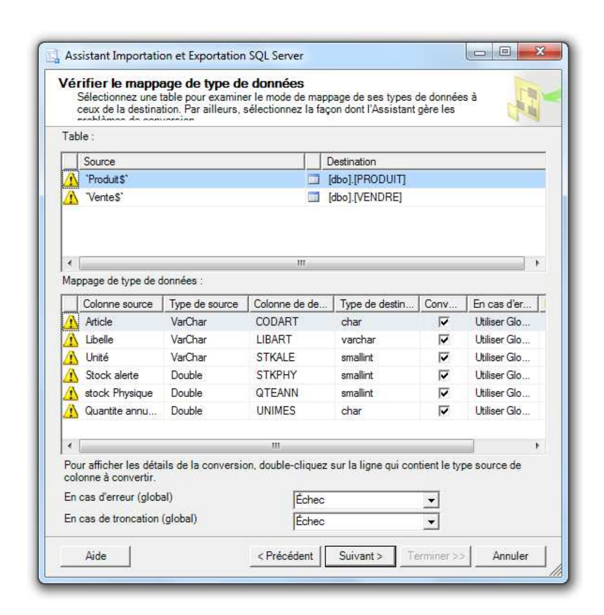
Sélectionnez les tables sources (onglet sous Excel) et les tables de destinations (votre base de données) correspondantes.

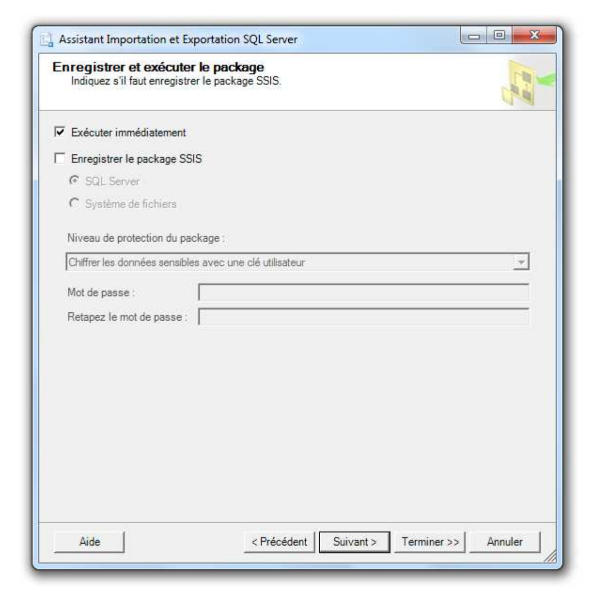


Vous aurez éventuellement à modifier les mappages notamment dans le cas où votre fichier Excel diffère de votre base.

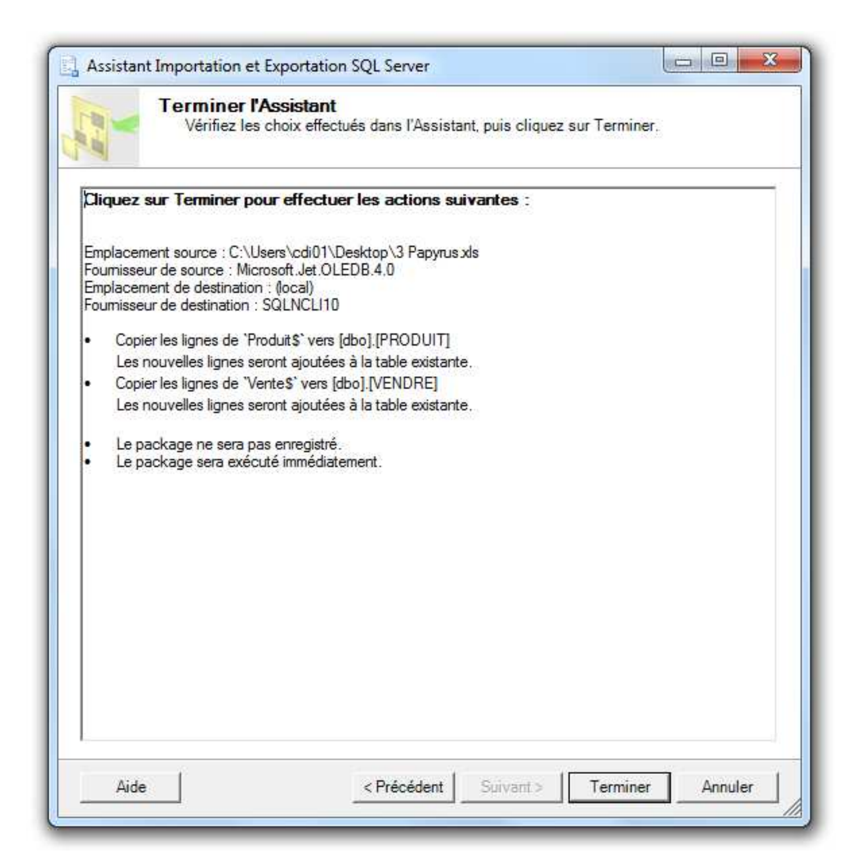


Cliquez toujours sur suivant…

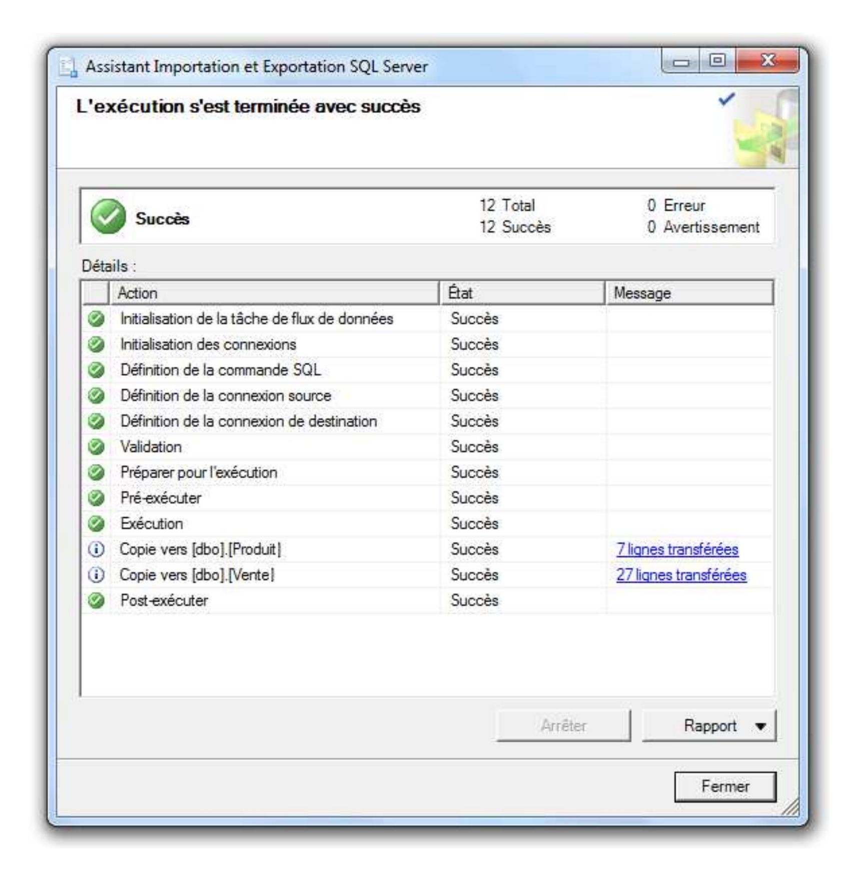




Puis cliquez sur terminer.



Normalement l'exécution doit se dérouler avec succès. Autrement, vérifier qu'il n'y a pas des lignes du fichier Excel en doublon avec votre table ou que les colonnes correspondent (types, nom…).



Les index

Si le but de l'index d'un livre est de nous permettre d'accéder plus rapidement au sujet qui nous intéresse dans ce livre, il en est de même pour les index dans la base de données, à la différence que ces index vont nous permettre de retrouver plus rapidement les données stockées dans la base.

Il existe plusieurs types d'index (index ou index-cluster, unique ou non unique…). Un index ordonné en clusters trie physiquement les lignes d'une table. Un index non ordonné en clusters s'appuie sur les informations d'emplacement de stockage contenues dans les pages d'index pour naviguer vers les pages de données (on parle de tri logique).

Une table ne peut posséder qu'un seul index ordonné en clusters, car ses lignes ne peuvent être stockées que dans un seul ordre physique. L'ordre physique des lignes de l'index et l'ordre des lignes de l'index sont identiques (il est conseillé de créer cet index avant tout autre).

Créer un index

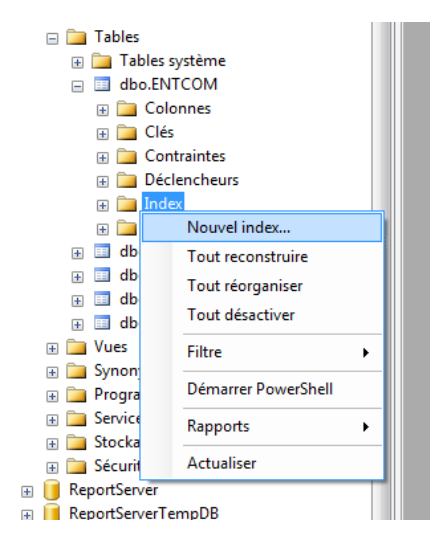
Un index peut être créé à n'importe quel moment, qu'il y ait ou non des données dans la table. Simplement, il est préférable de créer l'index après une importation majeure de données, pour éviter à avoir à le reconstruire par la suite, ce qui causera une perte conséquente de temps au niveau serveur.

L'instruction **CREATE INDEX** de Transact-SQL ou l'option « **Nouvel Index** » dans le menu contextuel du nœud Index de la table choisie de « Microsoft SQL Server Management Studio » peuvent être utilisés pour la création d'index.

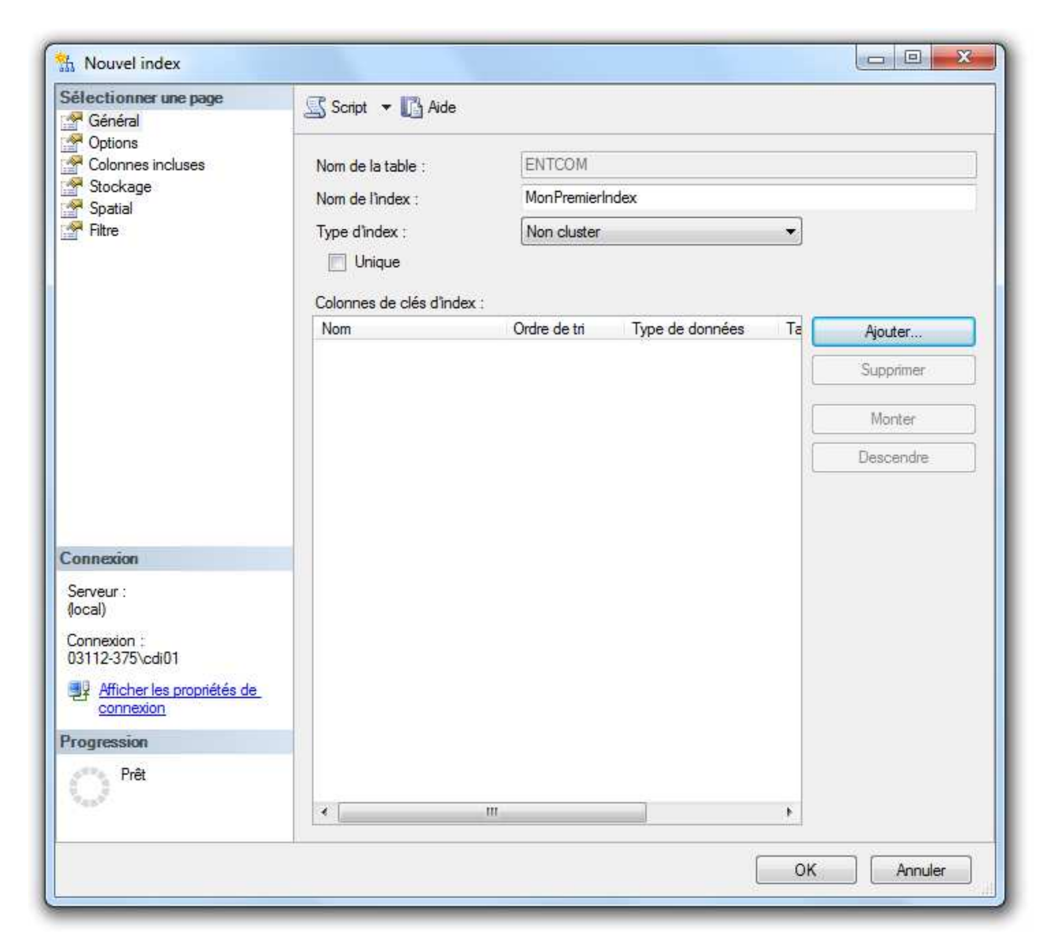
Prenons l'exemple suivant : Créez un index non-cluster sur la colonne NUMFOU de la table ENTCOM.

Par l'interface

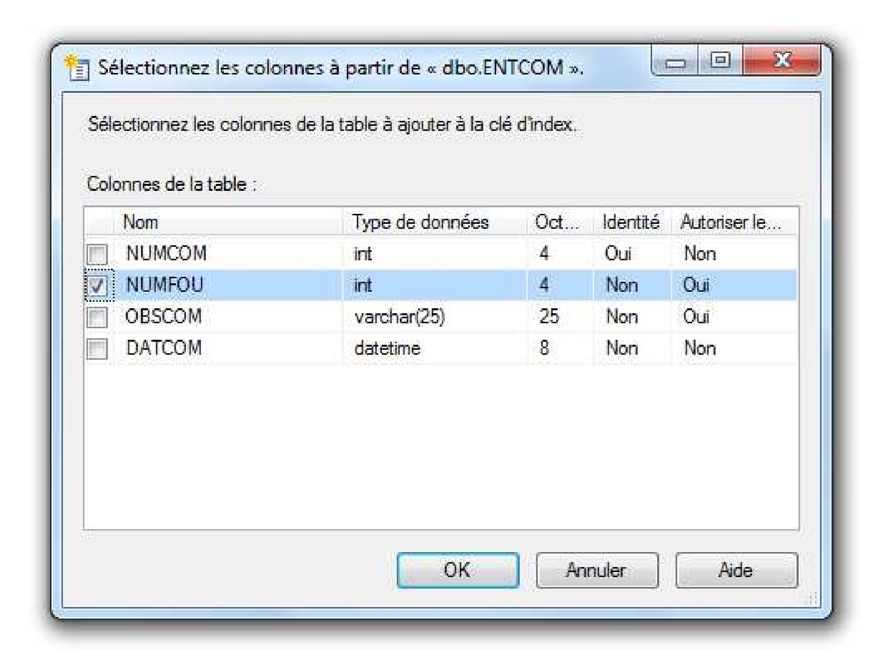
Au niveau du dossier Index de la table sur laquelle vous souhaitez créer votre index, cliquez droit puis « **Nouvel Index** ».



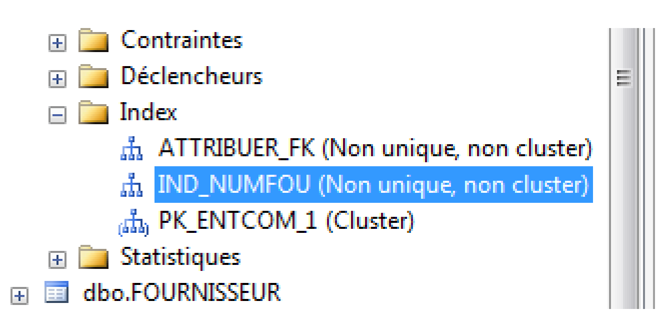
Donner un nom à votre index puis un type. Ici on précise le Type d'index « **Non Cluster** ». Cliquez sur « **Ajouter…** ».



Puis sélectionnez les colonnes sur lesquelles l'index va s'appliquer.



Cliquez sur « **Ok** ». Après actualisation, votre index figurera dans le dossier « **Index** ».



Par le code

Voici la syntaxe générale de création d'un index, que celui-ci soit ordonné ou non :

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX Nom_Index
ON Nom_Table (Nom_Colonnes1)
[INCLUDE (Nom_Colonnes2)]
[WITH]
[PAD_INDEX OFF], [FILLFACTOR = x],
[IGNORE_DUP_KEY = OFF], [DROP_EXISTING OFF],
[ONLINE = OFF], [STATISTICS_NORECOMPUTE = OFF]
[ON Nom_Groupe_Fichier]
```

Pour créer un index, nous nous servons de l'instruction **CREATE INDEX**. Les choix compris entre les mots clés **CREATE et INDEX** servent à décider si l'index doit être ordonné ou non, et s'il doit être unique. Avec la clause **ON**, on définit sur quelles colonnes porte l'index et avec la clause **INCLUDE**, on passe en paramètres plus de colonnes de la même table, afin que les requêtes n'aient qu'à chercher dans l'index lorsqu'elles s'exécutent. La clause **ON** en toute fin de lot permet de définir sur quel groupe de fichier nous allons placer notre index. On peut noter que si le groupe de fichier n'est pas précisé, l'index sera placé dans le groupe de fichier principal. À la suite de la clause **WITH**, nous trouvons les différentes options des index, qui par défaut sont toutes désactivées (**OFF**) mis à part **FILLFACTOR**. Voici le détail des services que proposent ces options :

- PAD_INDEX : Précise le niveau de remplissage du niveau non-feuille. Cette option n'est utilisable qu'avec **FILLFACTOR** dont la valeur est reprise.

- FILLFACTOR : Précise le pourcentage de remplissage des pages d'index au niveau feuille. La valeur par défaut est 0.

- IGNORE_DUP_KEY : Cette option autorise les entrées doubles dans les index de type UNIQUE.

- DROP_EXISTING : Précise que l'index existant doit être supprimé.

- ONLINE : Lorsque cette option est activée, les données de la table restent accessibles en lecture, lors de la création de l'index.

- STATISTICS_NORECOMPUTE : désactivée, cette option précise que les statistiques ne seront pas mises à jour.

Nous allons maintenant créer un index non ordonné sur notre base de données Entreprise, base de données dont nous nous servons pour tous nos exemples dans ce cours. Cette base est accessible en annexe. Prenons un exemple simple. Dans notre table Client de la base de données Entreprise, le mail des clients peut être à la valeur **NULL**. De plus, on remarque que le champ Mail_Client n'est occupé que pour 50% des cas, soit un cas sur deux. L'accès aux données est donc ralenti par le fait que la plupart des données sont à **NULL**. On peut donc créer un index non organisé de cette manière :

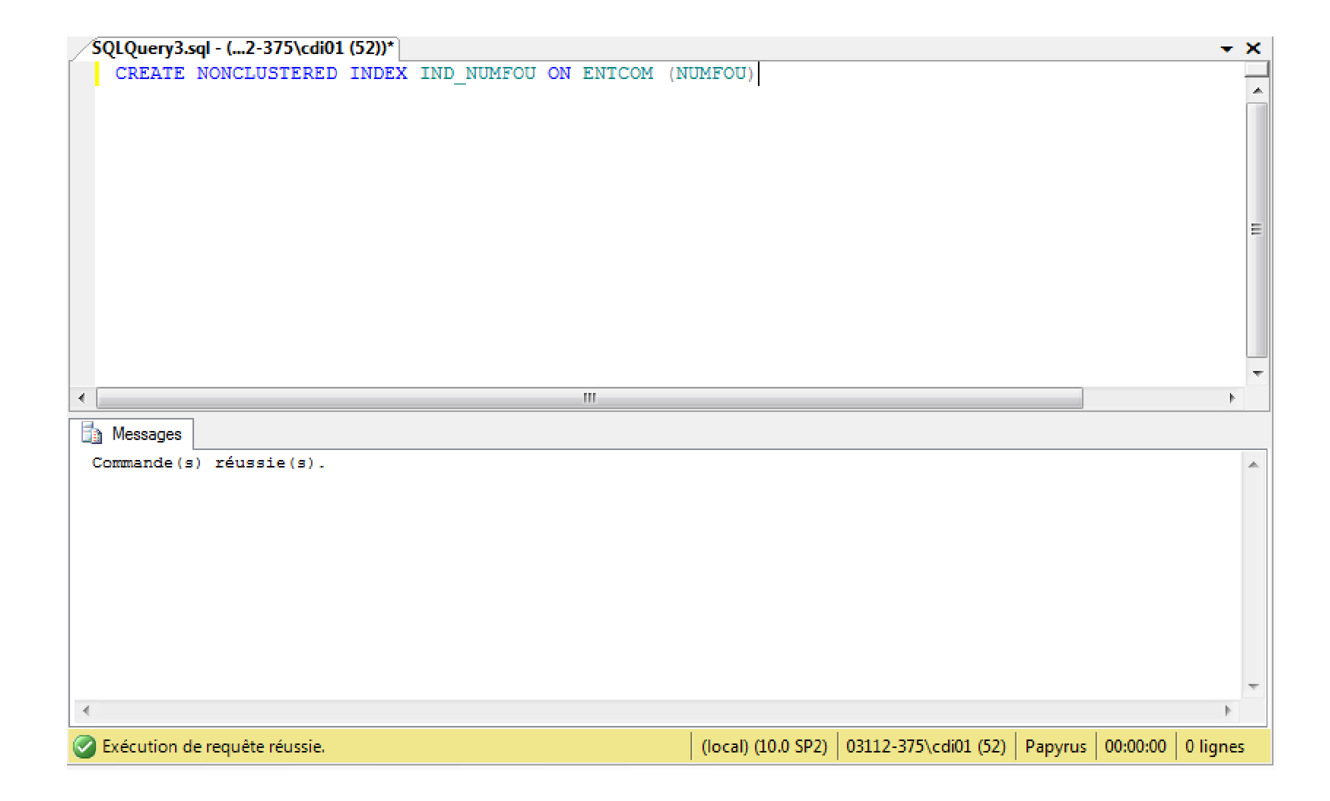
```
CREATE NONCLUSTERED INDEX Index_Mail
ON Client (Mail_Client)
INCLUDE (Id_Client)
WHERE Mail_Client IS NOT NULL
```

Avec ce segment de code, nous allons créer un index non organisé, qui s'appelle Index_Mail, sur la table Client, pour la colonne Mail_Client, à laquelle nous incluons Id_Client. Cet index va récupérer toutes les informations de la table Client pour laquelle Mail_Client n'est pas **NULL**, ce qui va nous permettre d'accélérer nos requêtes de façon conséquente, dans le cas où la masse de données sur la base est conséquente.

Un index est automatiquement créé lorsqu'une contrainte **PRIMARY KEY** ou **UNIQUE** est ajoutée à une table.

Dans la plupart des cas, il est préférable d'utiliser la contrainte **UNIQUE** plutôt qu'un index unique. L'instruction **CREATE INDEX** échouera s'il existe des doublons dans la colonne à indexer.

Pour revenir à notre exemple de départ « Créez un index non-cluster sur la colonne NUMFOU de la table ENTCOM », nous faisons simplement :

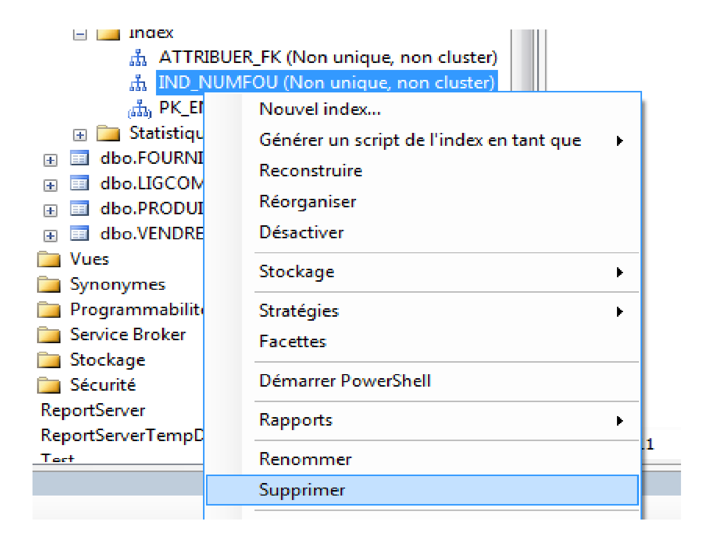


Supprimer un index

La suppression d'index peut avoir plusieurs origines. La plus fréquente est la suivante. Lorsqu'un index est trop coûteux en maintenance et qu'il n'offre pas de performances significatives sur les requêtes, il peut être préférable de le supprimer.

Par l'interface

Cliquez tout simplement droit et « **Supprimer** ».



Par le code

Comme chacun le sait désormais, le mot clé pour supprimer un objet de la base est le mot **DROP**. Nous allons encore une fois l'utiliser afin de pouvoir supprimer un index de la base. Voici la commande type de suppression d'un index :

```
-- DROP INDEX Nom_Index ON Nom_Table
DROP INDEX Index_Mail ON Client
```

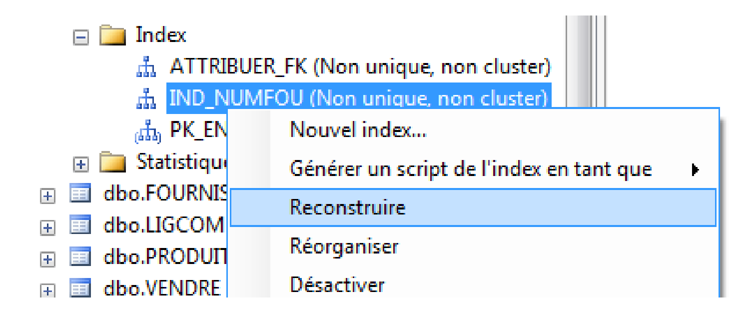
L'instruction **DROP INDEX** supprimera l'index.

Reconstruire un index

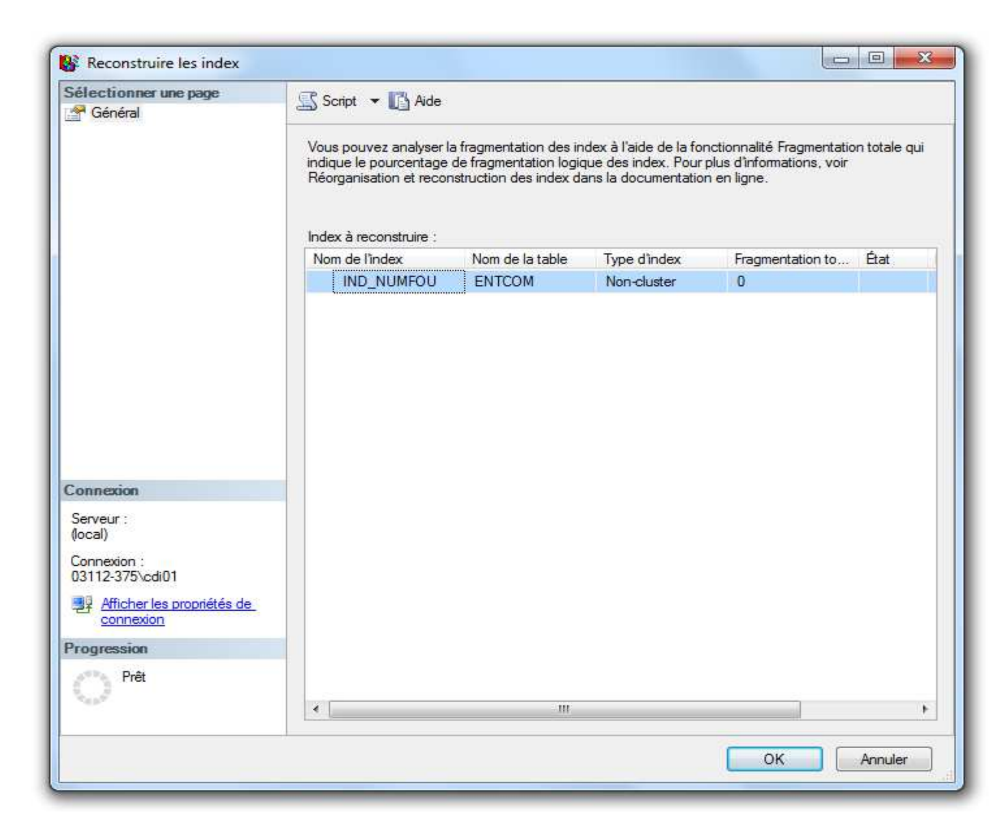
Nous allons voir que nous avons la possibilité de reconstruire les index directement par l'interface, mais également par le code.

Par l'interface

Cliquez droit sur l'index à reconstruire puis « **Reconstruire** ».



Cliquez sur « **Ok** ».



Par le code

La syntaxe de reconstruction est la suivante :

```
ALTER INDEX ( Nom_Index | ALL )
ON Nom_Table
REBUILD
[WITH]
[PAD_INDEX OFF], [FILLFACTOR = x],
[IGNORE_DUP_KEY = OFF], [DROP_EXISTING OFF],
[ONLINE = OFF], [STATISTICS_NORECOMPUTE = OFF]
[ON Nom_Groupe_Fichier]
```

À la suite de l'instruction **ALTER INDEX**, il est nécessaire de préciser le nom du ou des index à reconstruire, ou bien de préciser si l'on veut que tous les index soient reconstruits en précisant le mot clé **ALL**. La clause **ON** permet de préciser de quelle table sont originaires les éventuels index à reconstruire et le mot clé **REBUILD** permet de préciser que l'on veut reconstruire ces index. La dernière clause **WITH** permet quant à elle de préciser les caractéristiques des index à reconstruire. Toutes les options contenues dans le **WITH** fonctionnent de la même manière que pour la simple construction de l'index.

Les vues

On peut définir une vue comme étant une table dite virtuelle, qui a la même utilisation qu'une table, simplement une vue ne prend pas d'espace sur le disque, puisqu'elle ne stocke pas les données comme une table. Elle ne stocke que la requête d'extraction des données (**SELECT**).

Les vues sont un grand avantage quant à la gestion des données, vis-à-vis de l'utilisateur final. En effet, elles permettent tout d'abord de simplifier la structure des tables, qui peuvent parfois comporter une multitude de colonnes. On pourra alors choisir, en fonction de l'utilisateur, les colonnes dont il aura besoin, et n'inclure que ces colonnes dans notre vue. Une vue peut aussi permettre la réutilisation des requêtes. En effet, lorsque certaines requêtes sont souvent utilisées, une vue permettra de stocker cette requête et de l'utiliser plus facilement.

Une vue partitionnée est définie par une opération **UNION ALL** portant sur des tables membres structurées de façon identique, que ce soit sur une seule instance de SQL Server ou dans un groupe d'instances autonomes SQL Server.

Pour un partitionnement local des données, la méthode recommandée consiste à recourir à des tables partitionnées.

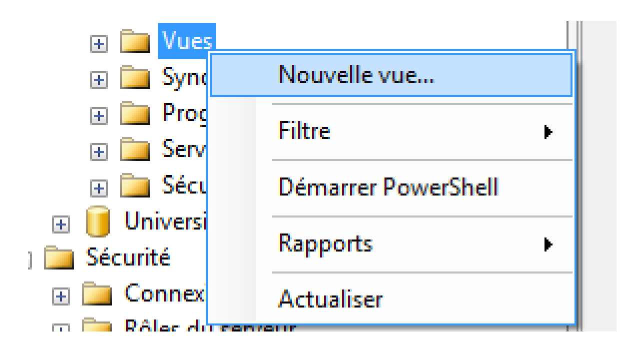
Si les tables d'une vue partitionnée se trouvent sur des serveurs différents, toutes les tables impliquées dans la requête peuvent être analysées en parallèle, ce qui accroît les performances pour cette requête.

Création d'une vue

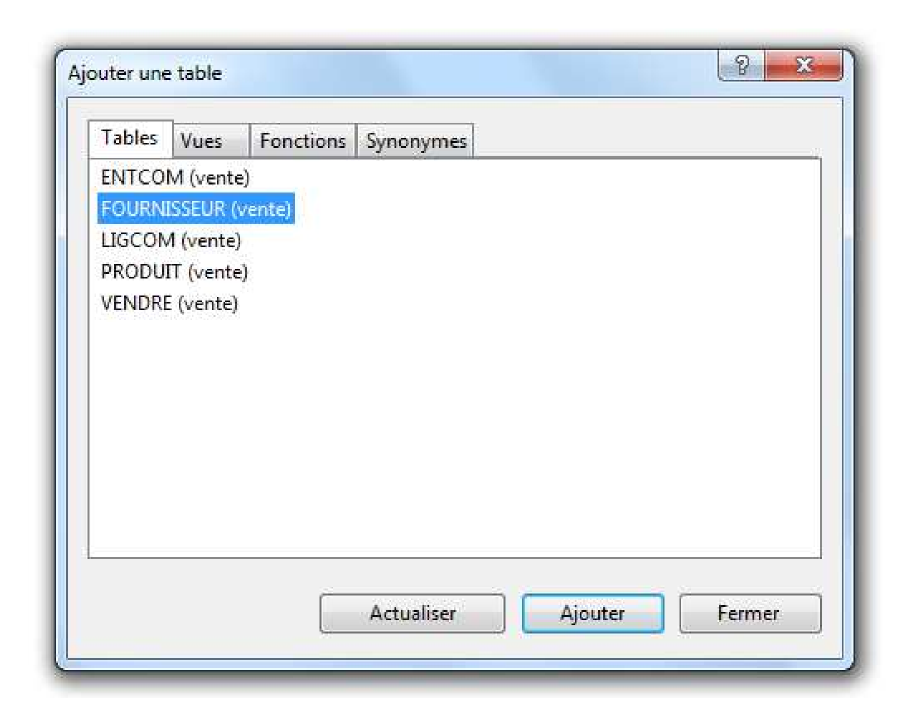
Une vue permet de stocker une requête prédéfinie sous forme d'objet de base de données, pour une utilisation ultérieure. Nous allons voir les deux possibilités existantes pour créer une vue.

Avec l'interface

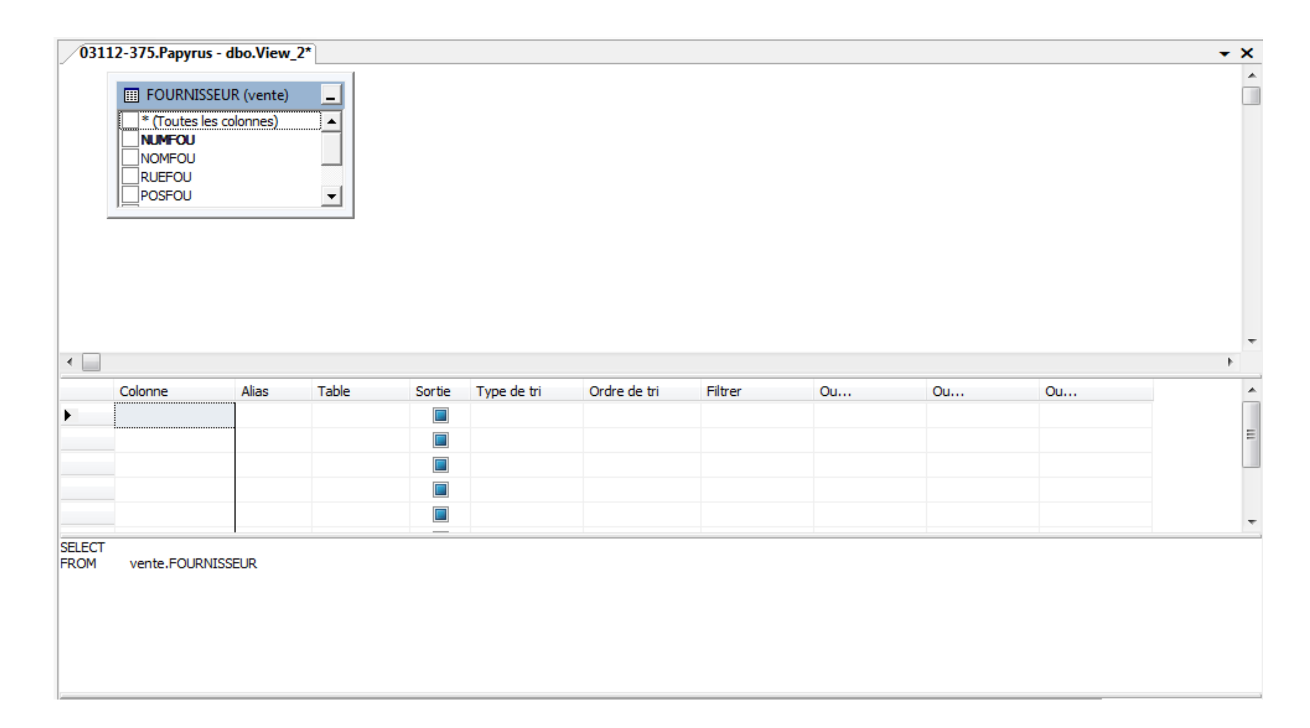
Il est très intuitif de créer une vue grâce à SSMS. Pour de faire, de manière graphique, il vous suffit de cliquer droit sur le sous dossier « **Vues** » dans votre base de données, affichée dans l'explorateur d'objet. Après avoir cliqué droit, sélectionnez « **Nouvelle vue…** »



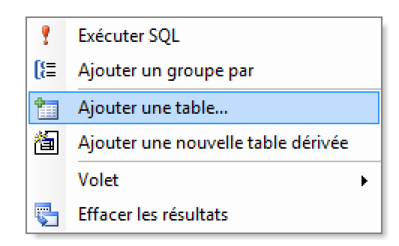
Après avoir cliqué sur « **Nouvelle vue…** », les deux fenêtres suivantes apparaissent au sein même de SSMS. La première vous aidera à sélectionner des tables sur lesquelles la vue portera. La seconde vous permet de sélectionner les colonnes à utiliser et construire votre requête.



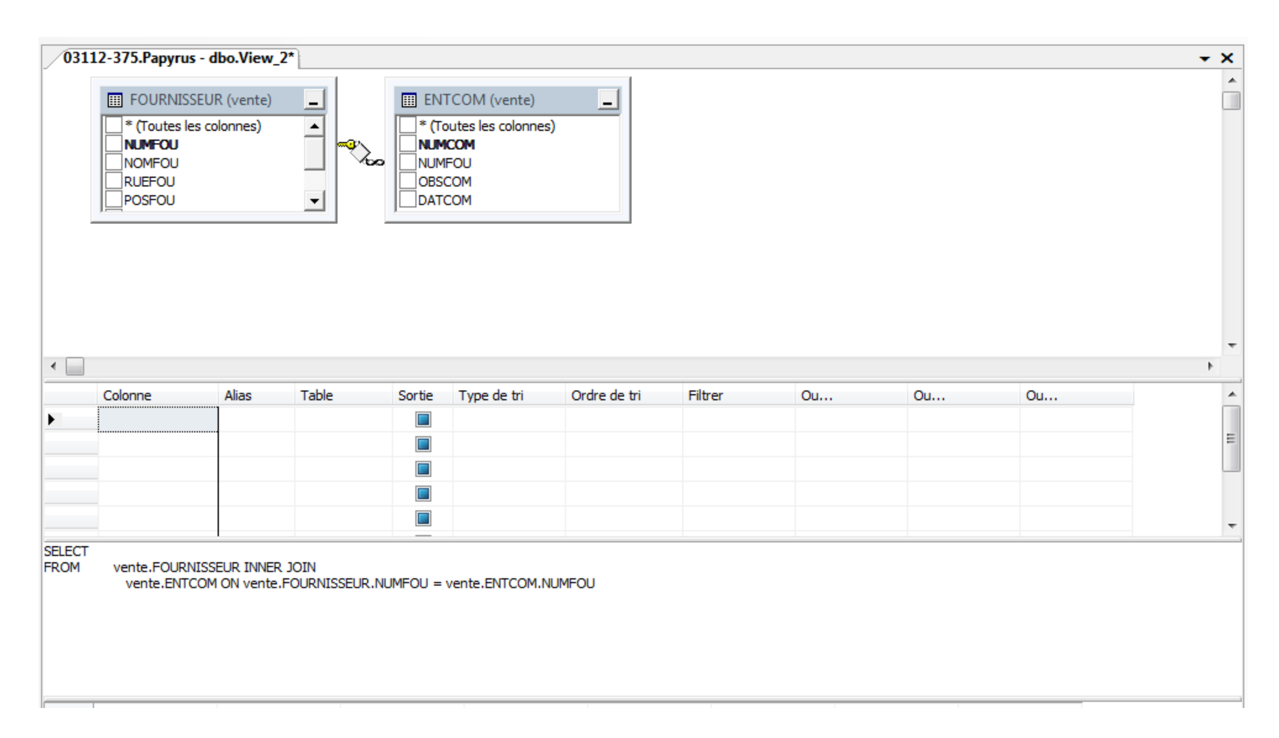
Pour ajouter une table, cliquez sur le nom de la table voulue, et sélectionnez « **Ajouter** ».



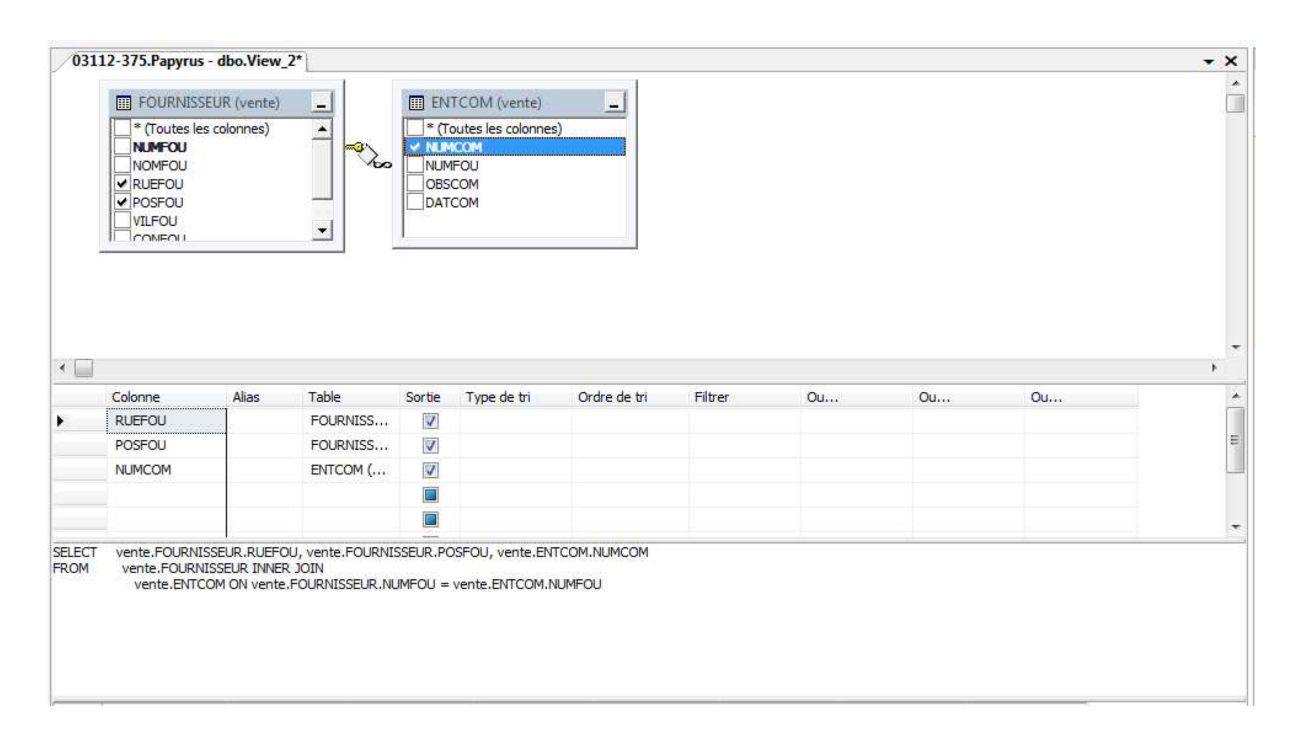
Vous pouvez en ajouter plusieurs par simple cliquez droit puis « **Ajouter une table…** ».



On remarquera que les tables, aussitôt sélectionnées, sont modélisées dans la partie supérieure de la seconde fenêtre.



Pour sélectionner les colonnes à mettre dans votre vue, cochez les cases correspondant à vos colonnes dans les tables modélisées dans la partie supérieure de la fenêtre. Lorsque l'on coche des cases, on peut remarquer que le nom de ses colonnes est ajouté dans la partie centrale de la fenêtre.

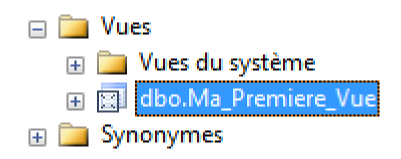


On peut alors modifier le type de tri, l'ordre, ou encore le filtre de cette vue en changeant les caractéristiques dans la même partie. Toutes les actions effectuées permettront de générer le code de la vue. Voici un exemple :

```
SELECT    vente.FOURNISSEUR.RUEFOU, vente.FOURNISSEUR.POSFOU, vente.ENTCOM.NUMCOM
FROM      vente.FOURNISSEUR INNER JOIN
            vente.ENTCOM ON vente.FOURNISSEUR.NUMFOU = vente.ENTCOM.NUMFOU
```

Lorsque vous avez fini de concevoir votre vue, faites cliquer droit sur l'onglet de la fenêtre dans SSMS, et choisissez « **Enregistrer** ». Donnez-lui un nom. Vous venez de créer votre vue.

Après actualisation, vous pourrez alors retrouver votre vue dans le sous-dossier de votre explorateur d'objet, comme présenté dans l'image ci-dessous :



Création d'une vue avec du code T-SQL

La syntaxe de création d'une vue avec du code T-SQL est simple. On utilisera l'instruction **CREATE**, comme pour toute création d'objets dans une base de données.

```
CREATE VIEW Nom_Vue
[options1]
AS requête [options2]
```

Nous utilisons l'instruction **CREATE VIEW**, auquel nous associons le nom que nous voulons lui donner. Le mot clé **AS** indique que nous allons spécifier la requête **SELECT** qui va nous permettre de sélectionner les colonnes d'une ou plusieurs tables, afin d'en copier les propriétés dans la vue que nous créons. Il est bon de préciser que des clauses existantes pour une instruction **SELECT** classique ne conviendra pas pour une instruction **SELECT** servant à créer nos vues. Ces instructions ne doivent pas être autres que l'instruction **SELECT**, et les clauses **FROM** et **WHERE**. Concernant les options 1 et 2 : L'option 1 correspond aux options suivantes : **WITH ENCRYPTION**, **WITH SCHEMABINDING**, **WITH VIEW_METADATA** ; et l'option 2 correspond à l'option suivant : **WITH CHECK OPTION**. Découvrons les actions de chacune de ces options sur notre vue.

WITH ENCRYPTION : Permet de crypter le code dans les tables système. Attention : personne ne peut consulter le code de la vue, même pas son créateur. Lors de la modification de la vue avec l'instruction **ALTER VIEW**, il sera nécessaire de préciser de nouveau cette option pour continuer à protéger le code de la vue.

WITH SCHEMABINDING : Permet de lier la vue au schéma. Avec cette option, il est impératif de nommer nos objets de la façon suivante : schéma.objet.

WITH VIEW_METADATA : Permet de demander à SQL Server de renvoyer les métadonnées correspondantes à la vue, et non celles qui composent la vue.

WITH CHECK OPTION : Permet de ne pas autoriser l'insertion ni la modification des données ne correspondant pas aux critères de la requête.

Voici un exemple :

```
CREATE VIEW Ma_Premiere_Vue
WITH ENCRYPTION
AS SELECT Id_Client, Nom_Client
FROM dbo.Client
```

Cet exemple permet de créer une vue dont le nom est Ma_Première_Vue, avec l'option **WITH ENCRYPTION**, et cette vue contiendra les colonnes Id_Client_archive et Id_Commande_ archive de la table Archive.

<u>Exemple 1</u> : Créez la vue VentesI100 correspondant à la requête : « Afficher les ventes dont le code produit est le I100 » (Affichage de toutes les colonnes de la table Vente)

```
-- VentesI100
CREATE VIEW VentesI100
-- Affichage de toutes les colonnes de la table Vente
AS SELECT *
-- Afficher les ventes (donc de la table Vente)
FROM vente.VENDRE
-- Où le code produit est I100
WHERE CODART = 'I100';
```

Exemple 2 : À partir de VentesI100, créer la vue VentesI100Grobrigan (toutes les ventes concernant le produit I100 et le fournisseur 00120).

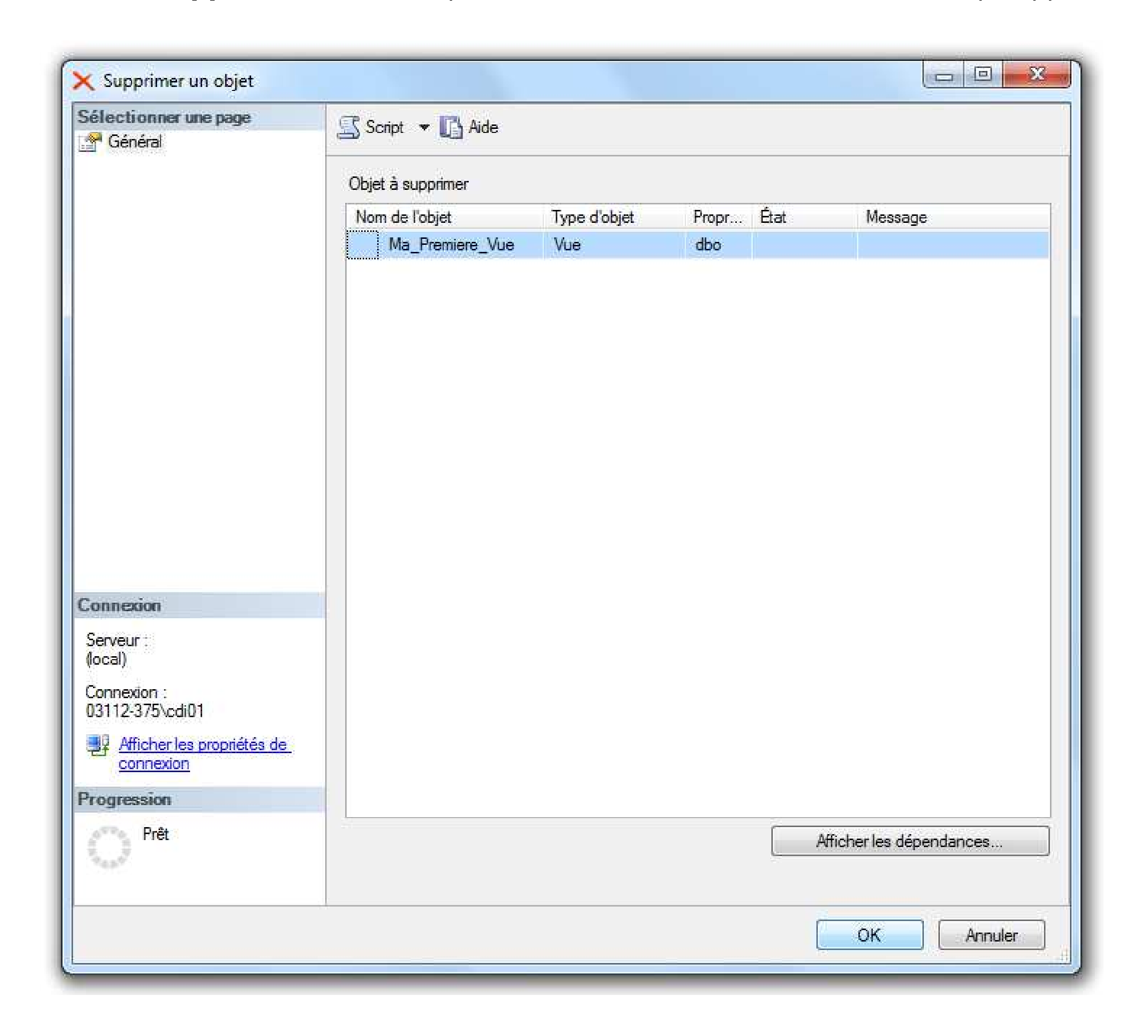
```
-- VentesI100Grobrigan
CREATE VIEW VentesI100Grobrigan
AS SELECT *
FROM VentesI100
WHERE NUMFOU = 00120;
```

Suppression d'une vue

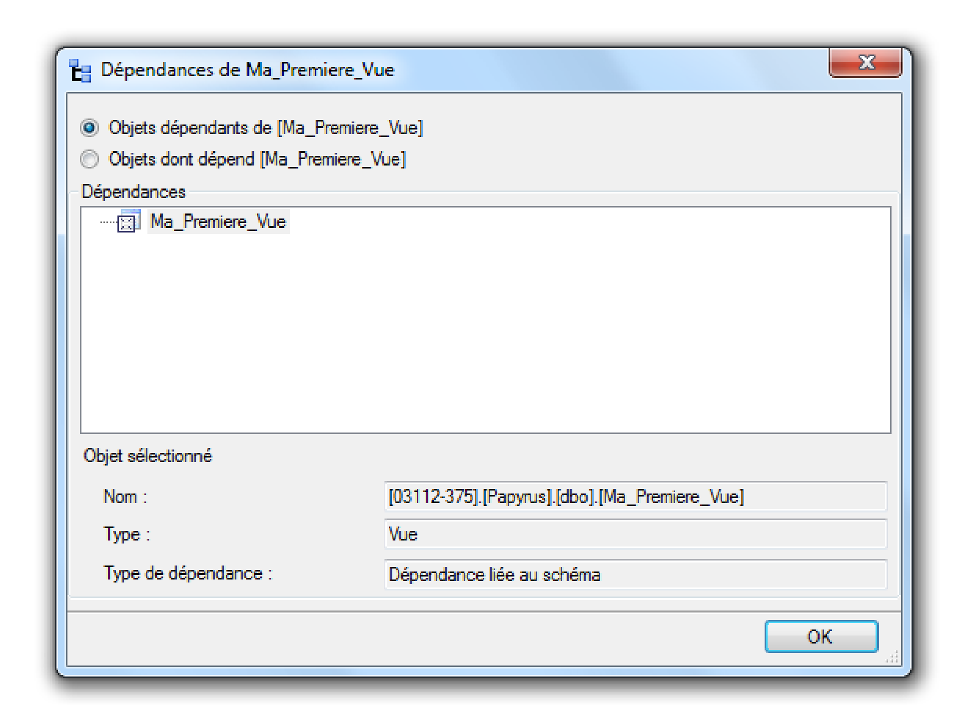
Pour supprimer une vue, nous pouvons le faire via l'interface ou par le code.

Suppression d'une avec l'interface

Avec SSMS, il vous suffit de cliquer droit sur la vue dans l'explorateur d'objet et de sélectionner « **Supprimer** » et de cliquer sur « **Ok** » dans la nouvelle fenêtre qui apparaît.



Grâce au bouton, « **Afficher les Dépendances…** », Il est possible de mettre en évidence les dépendances existantes entre les objets de la base et la vue sélectionnée. Une nouvelle fenêtre s'affiche à l'écran, dans laquelle vous pourrez choisir les différents types de dépendances.



Il vous suffit juste de savoir si vous voulez afficher les objets dépendants de la vue ou ceux dont dépend la vue en question.

Avec du code T-SQL

La structure de suppression d'une vue est la même que pour tout objet de la base de données. Elle est la suivante :

```
DROP VIEW nom_vue
```

Les vues indexées

Les vues indexées ont un unique objectif, comme tout autre objet indexé dans la base de données : améliorer les performances de nos requêtes. Celles-ci sont sûrement les objets de la base offrant le plus de gain de performance dans SQL Server. On se sert le plus souvent de ce type de vues, sur des bases de données OLAP, c'est-à-dire pour des données qui vont être le plus souvent en lecture, et très peu mises à jour. Ces index sont en particulier pratiques pour des requêtes nécessitant des jointures et des agrégations. La vue indexée a pour effet de matérialiser les données. On prend le résultat de la requête et on le stocke dans l'index. On met ensuite à jour l'index en fonction des modifications sur la table de base. La création d'un index sur une vue se fait de la même manière que pour une table, simplement, un index sur une vue présente de caractères propres de fonctionnement et de comportement que nous avons commencé à présenter. Pour une version Entreprise de SQL Server, dès lors que votre index est créé, le moteur de base de données utilisera celui-ci. En revanche, pour les autres versions de SQL Server, il faut préciser si l'on veut utiliser l'index. La méthode est :

```
SELECT * FROM vue_Client WITH(NOEXPAND)
```

La clause **WITH(NOEXPAND)** spécifie qu'aucune vue indexée n'est étendue pour permettre d'accéder aux tables sous-jacentes lorsque l'optimiseur de requête traite la requête. L'optimiseur de requête traite la vue comme une table avec un index cluster. **NOEXPAND** s'applique uniquement aux vues indexées. Des contraintes sont à noter. En effet, il n'est possible de créer un index sur une vue que si les conditions suivantes sont rassemblées :

- Lors de la création de la vue, les options **ANSI_NULLS** et **QUOTED IDENTIFIER** doivent être sur **ON**.
- **ANSI_NULL** doit être **ON** lors de la création de toutes les tables référencées dans la vue sur laquelle sera créé l'index.
- La vue ne doit pas faire référence à d'autres vues.
- Toutes les tables référencées doivent appartenir à la même base et au même propriétaire.
- La vue doit être créée avec l'option **WITH SCHEMABINDING**.

Le premier index créé sur une vue doit être de type cluster unique. Par la suite il est possible de construire des index non cluster.

Pour faire un résumé rapide, les vues indexées sont pratiques et efficaces dans le cas où une vue possède une quantité remarquable de jointures et d'agrégations, et que ces données sont surtout en lecture sur la base. Enfin, pour une création sans erreurs de l'index, certaines conditions doivent être respectées.

Des index peuvent être créés sur des vues. Une vue indexée stocke l'ensemble des résultats d'une vue dans la base de données. En raison de leur faible temps d'extraction, les vues indexées peuvent être utilisées pour améliorer les performances d'une requête.

Une vue indexée est créée en implémentant un index **UNIQUE CLUSTERED** sur la vue. D'autres index peuvent être créés.

Une vie indexée renvoie les modifications effectuées sur les données dans les tables de base. L'index **UNIQUE CLUSTERED** est mis à jour à mesure que les données sont modifiées.

Une vue peut être modifiée par **ALTER VIEW**, et supprimée par **DROP VIEW**.

Gestion des schémas

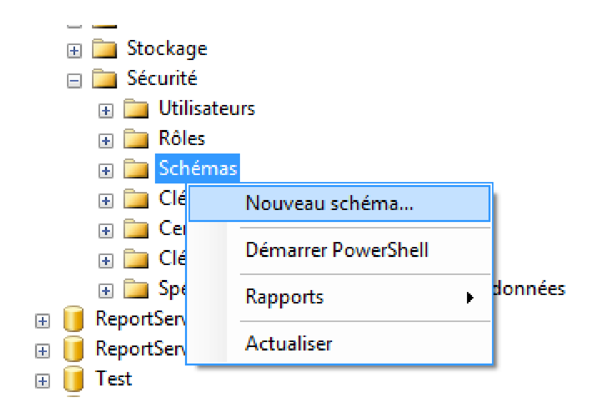
Un schéma est un ensemble logique d'objets à l'intérieur des bases de données sur le serveur. Leur but est de faciliter, entre autres, l'échange de données entre les utilisateurs, sans pour autant affecter la sécurité. Concrètement, les schémas permettent une gestion plus aisée des privilèges d'utilisation des objets. Comme nous l'avons vu précédemment, un utilisateur est mappé sur un schéma dès sa création, obligatoirement. Si toutefois, aucun nom de schéma n'est précisé, alors l'utilisateur sera mappé sur **dbo** par défaut. Pour expliquer simplement le fonctionnement des schémas, prenons un exemple : un utilisateur est mappé sur un schéma nommé RU. Pour requêter sur les objets de la base, il pourra écrire directement le nom seul de cet objet si celui-ci est compris dans le schéma sur lequel il est mappé. Dans le cas contraire, l'utilisateur devra préciser le schéma de l'objet, le nom de l'objet et à ce moment-là, SQL Server cherchera si ledit utilisateur possède les droits d'utiliser l'objet auquel il tente d'accéder. On peut retenir que les schémas servent essentiellement à faciliter le partage d'information entre les utilisateurs mappés sur ce schéma.

Créer un schéma

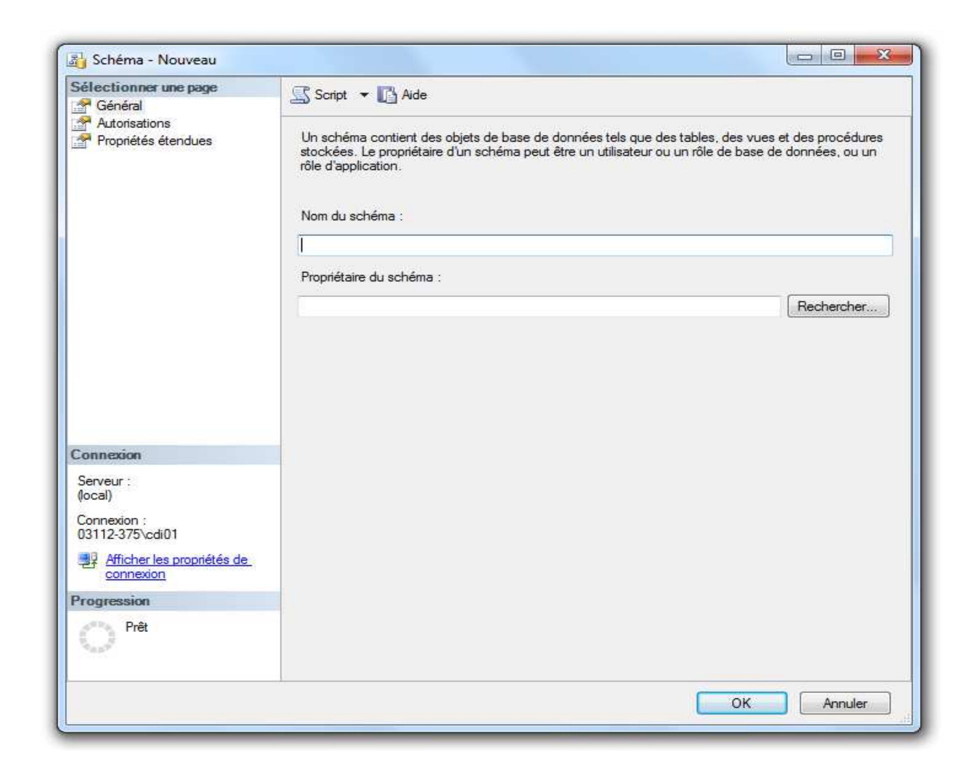
Vous pouvez créer un schéma directement à partir de l'interface où simplement à partir du code.

Avec l'interface

Pour créer un schéma grâce à SSMS, il vous suffit de déployer tous les nœuds jusqu'à **sécurité**, d'afficher le menu contextuel (cliquez droit) du nœud **schéma** et de sélectionner « **Nouveau schéma** ». Une nouvelle fenêtre apparaît.



Donnez un nom à votre schéma et configurez à votre guise.



Avec du code T-SQL

Voici la syntaxe de création d'un schéma de base de données :

```
CREATE SCHEMA nomschema
```

```
AUTHORIZATION nomproprietaire
options
```

Pour créer un schéma de base de données, nous nous servons de l'instruction **CREATE SCHEMA**. Comme tout objet dans une base de données, un schéma doit avoir un nom unique dans la famille des schémas. Il est donc nécessaire de préciser un nom derrière cette instruction. La clause instruction va nous permettre en revanche de préciser l'utilisateur de base de données qui sera propriétaire du schéma. Pour finir, ce que nous avons représenté par option représente l'espace dans lequel nous pouvons directement faire la définition de nos tables, vues et privilèges rattachés à nos tables.

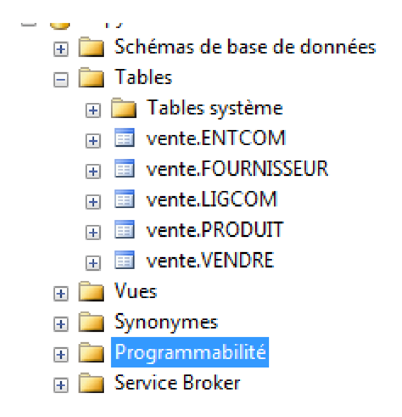
Ici nous créons un schéma vente.

```
CREATE SCHEMA vente
```

Nous appliquons ce schéma vente au tables suivantes :

```
ALTER SCHEMA vente TRANSFER dbo.ENTCOM
ALTER SCHEMA vente TRANSFER dbo.FOURNISSEUR
ALTER SCHEMA vente TRANSFER dbo.LIGCOM
ALTER SCHEMA vente TRANSFER dbo.PRODUIT
ALTER SCHEMA vente TRANSFER dbo.VENDRE
```

Après actualisation :



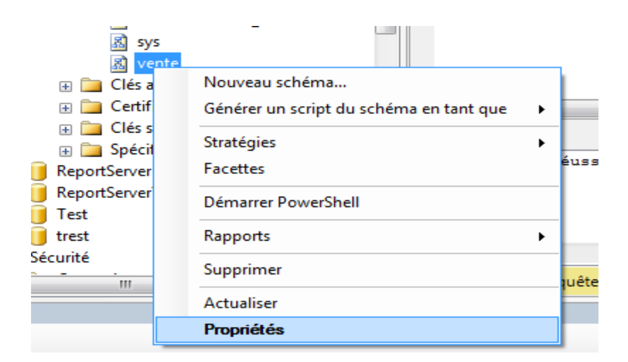
Modification d'un schéma

Vous pouvez modifier un schéma directement à partir de l'interface où à partir du code T-SQL.

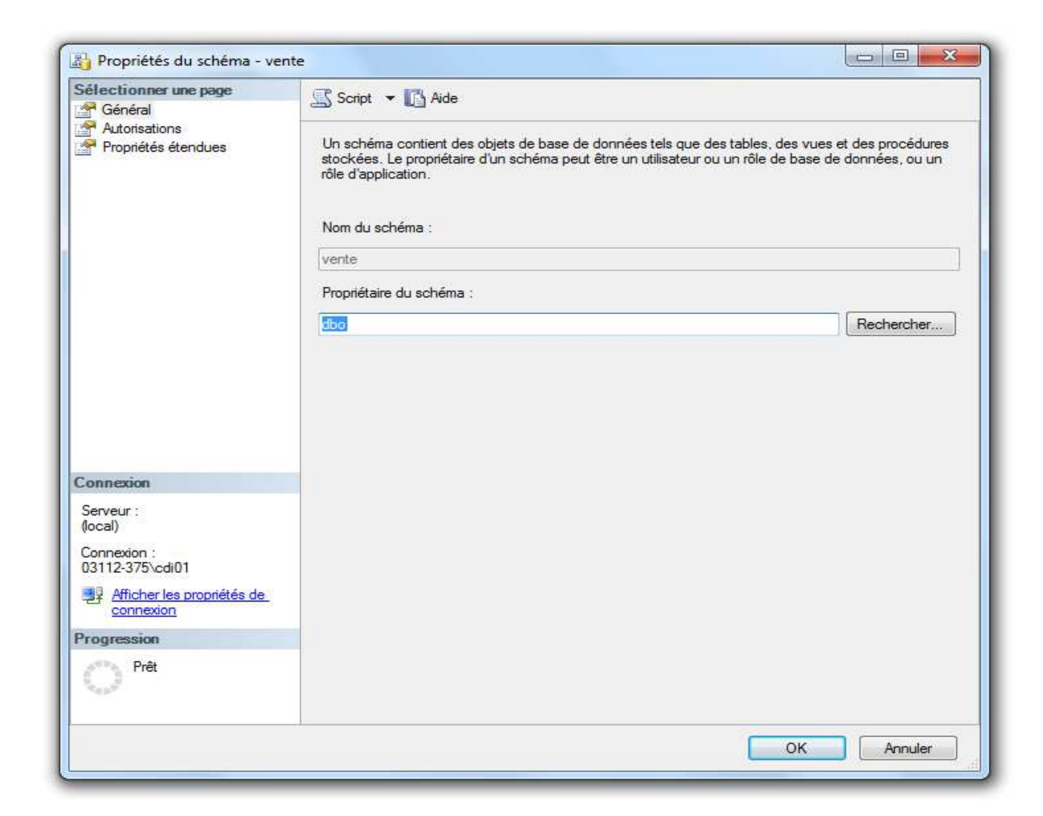
Avec l'interface

Avec l'interface graphique, la manipulation est la même que pour la création d'un schéma. Cependant, il faudra veiller à ne pas créer un nouveau schéma, mais à se rendre dans les propriétés du schéma que nous voulons modifier. Il faut en revanche prendre en compte que l'on ne peut pas changer le nom du schéma, seulement les tables, vue qui y sont contenues, les autorisations et le propriétaire du schéma.

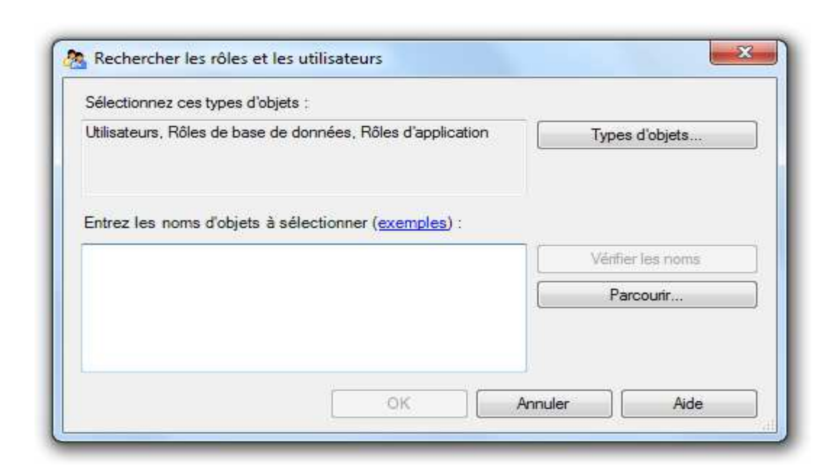
Par exemple, nous souhaitons désigner l'utilisateur « **Util5** » comme propriétaire du schéma vente : Cliquez droit sur le schéma vente puis « **Propriétés** ».



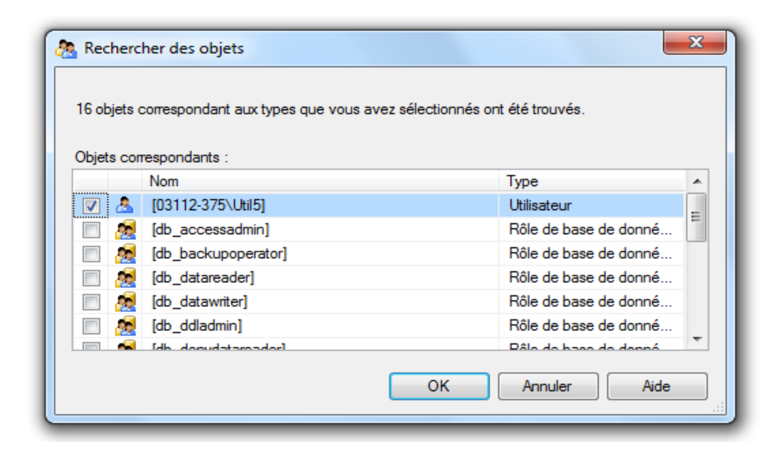
Cliquez sur « **Rechercher…** »



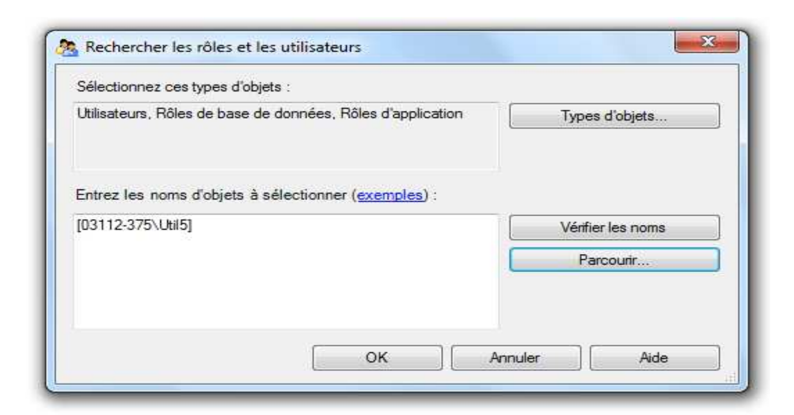
Puis cliquez sur « **Parcourir** ».



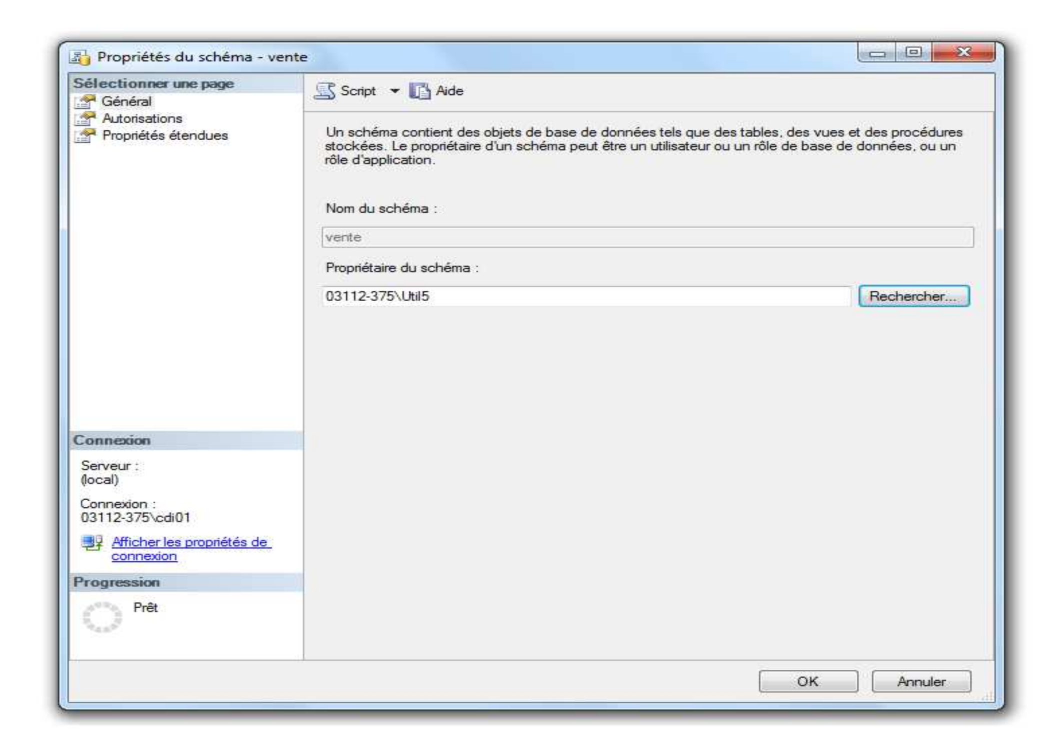
Sélectionnez l'utilisateur concerné puis cliquez sur « **Ok** ».



Cliquez de nouveau sur « **Ok** ».



« **Util 5** » est maintenant propriétaire du schéma.



Avec du code

Pour modifier un schéma avec du code, nous allons utiliser cette syntaxe :

```
ALTER SCHEMA nomschema
TRANSFERT nomobjet
```

Pour modifier un schéma, nous allons utiliser l'instruction **ALTER SCHEMA** suivie du nom du schéma à modifier. Après la clause **TRANSFERT**, on peut spécifier les divers objets à déplacer dans le schéma. Il faut savoir que le nom de ces objets doit être au format suivant : AncienSchema.NomObjet.

Exemple : Affecter « **Util5** » (du domaine « 03112-375 ») comme propriétaire du schéma vente.

```
ALTER AUTHORIZATION ON SCHEMA::vente TO [03112-375\Util5];
GO
```

Suppression d'un schéma

Vous pouvez supprimer un schéma directement à partir de l'interface où à partir du code T-SQL.

Avec l'interface

Pour supprimer un schéma de base de données avec SSMS, rendez-vous sur le nœud schéma de la base de données ou est mappé le schéma. Déployez ce nœud pour laisser apparaître les schémas existants. Faites un clic droit sur le schéma à supprimer et sélectionnez Supprimer.

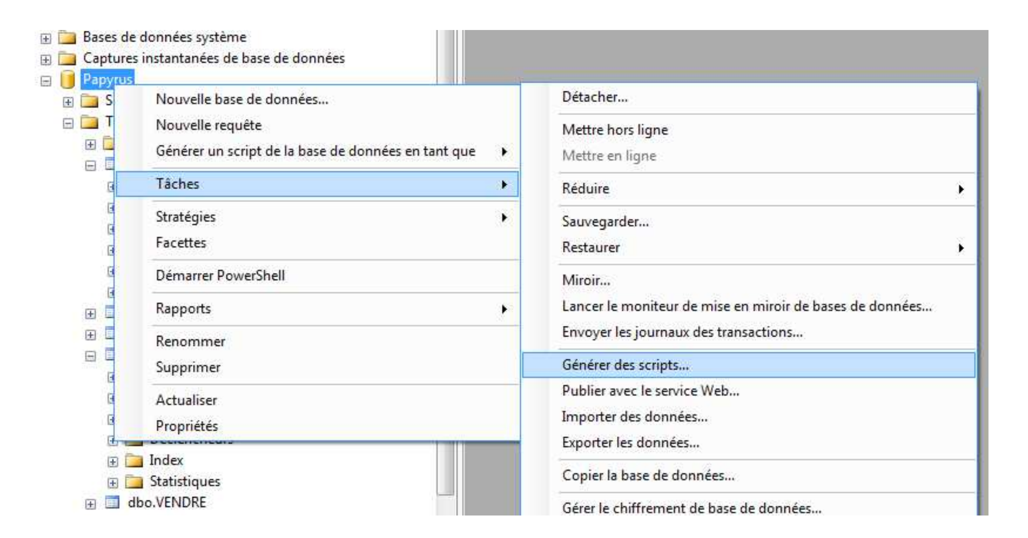
Avec du code

Pour supprimer un schéma, nous allons utiliser l'instruction **DROP SCHEMA**, avec la syntaxe suivante :

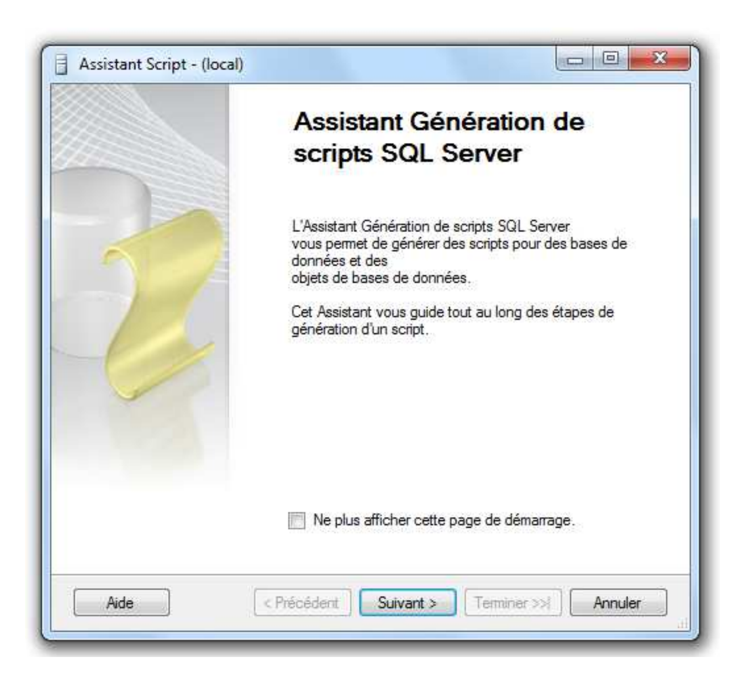
```
DROP SCHEMA nomschema
```

Générer des scripts

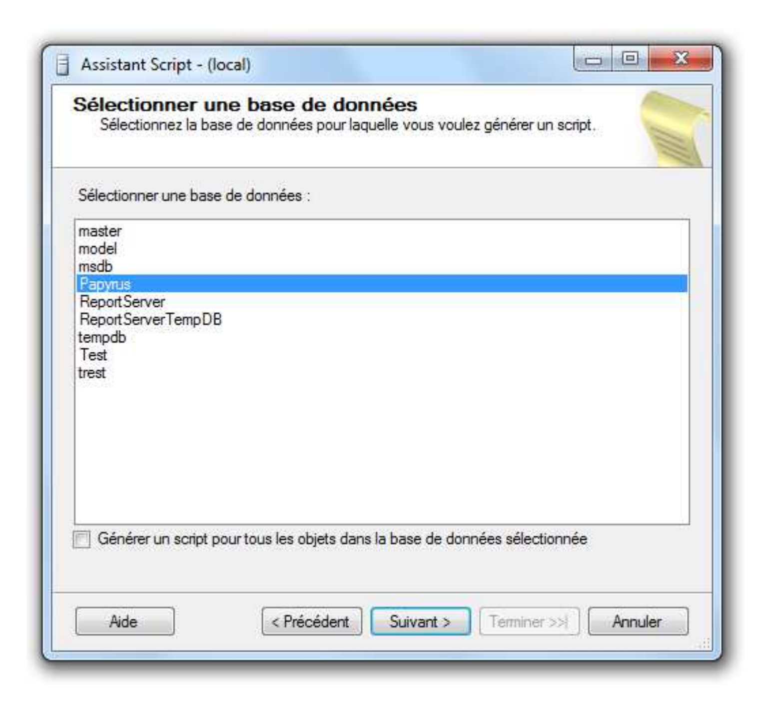
Pour générer le script de la base (dans nos cas pratiques : Papyrus), cliquez droit sur la base concernée, sélectionner « **Tâches** » puis « **Générer des scripts** ».



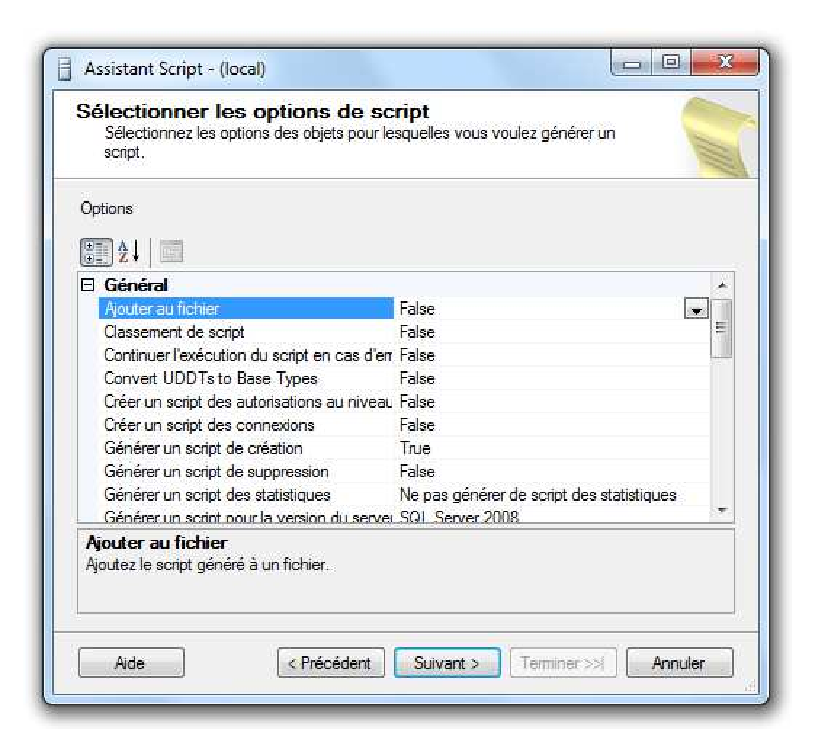
Laissez-vous guider par l'assistant.



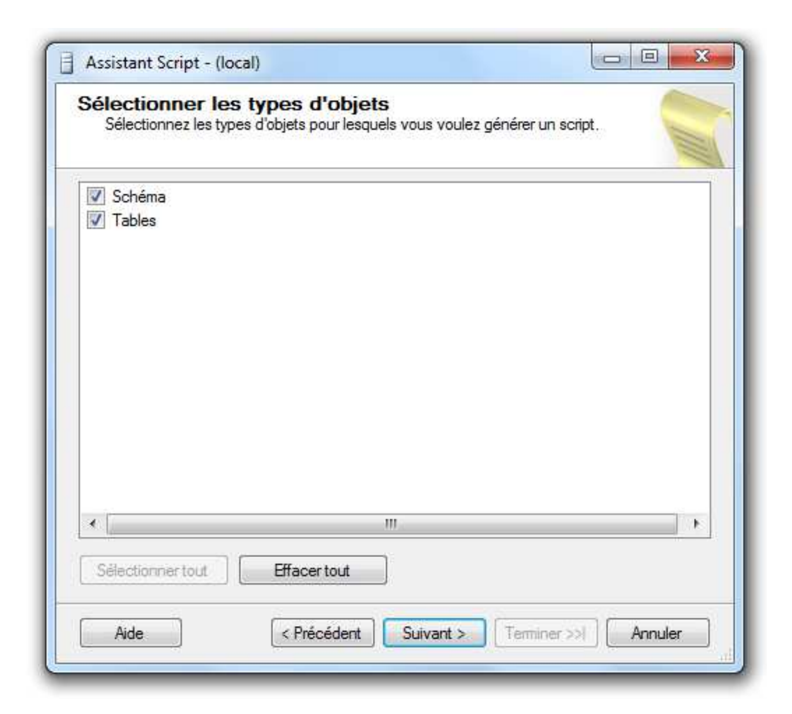
Sélectionnez votre base de données Papyrus.



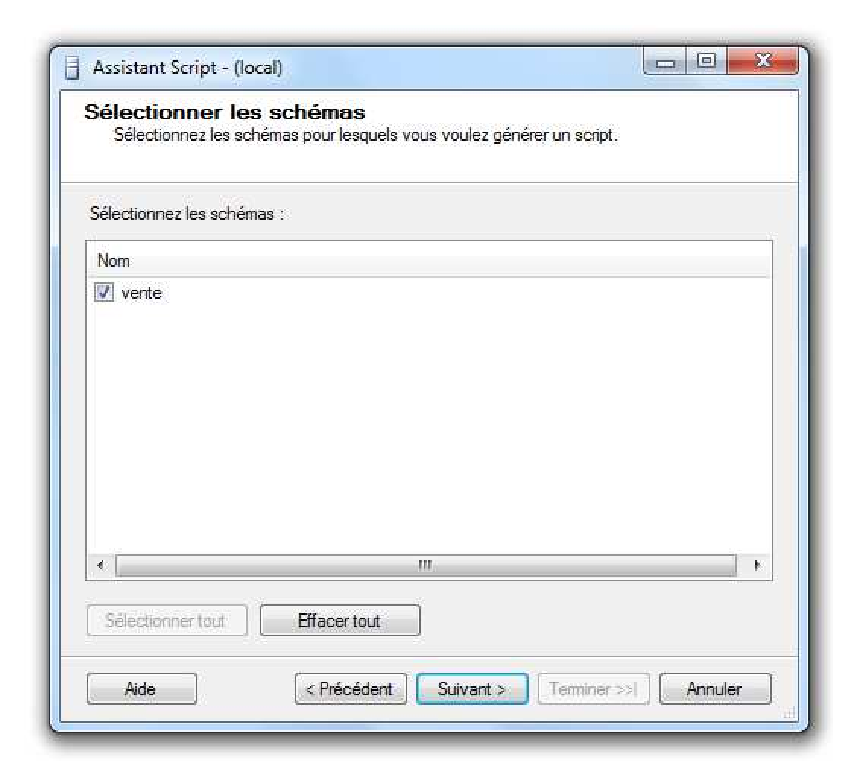
Conservez les valeurs par défaut.



Cochez les cases « **Schéma** » et « **Tables** ».

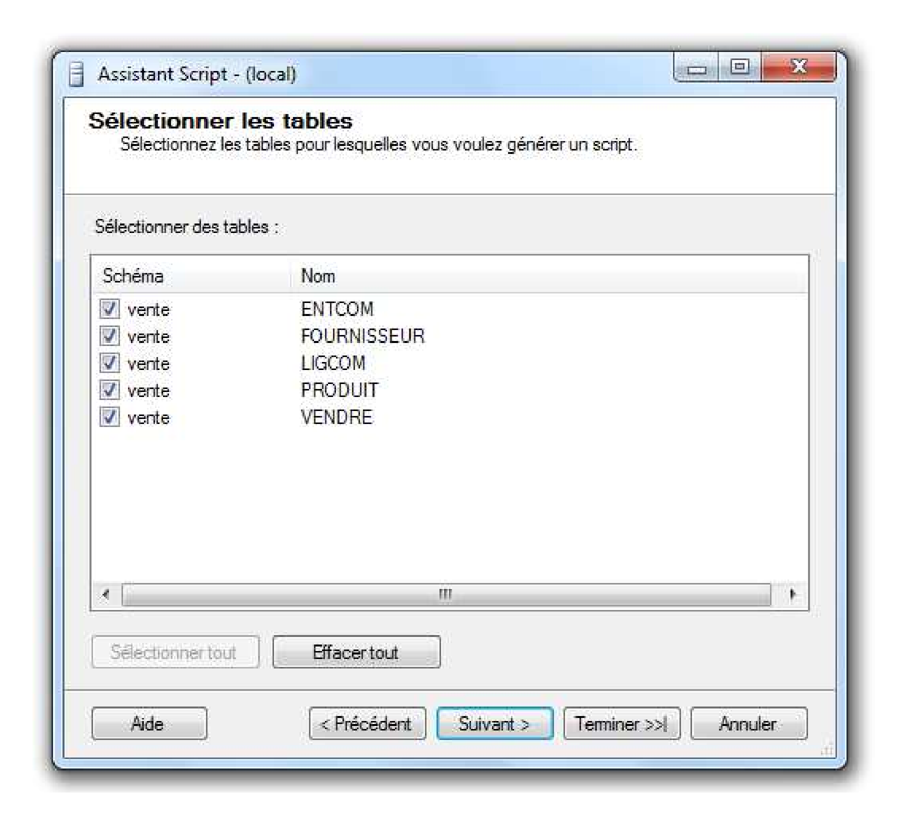


Cochez la case vente correspondante à votre schéma.

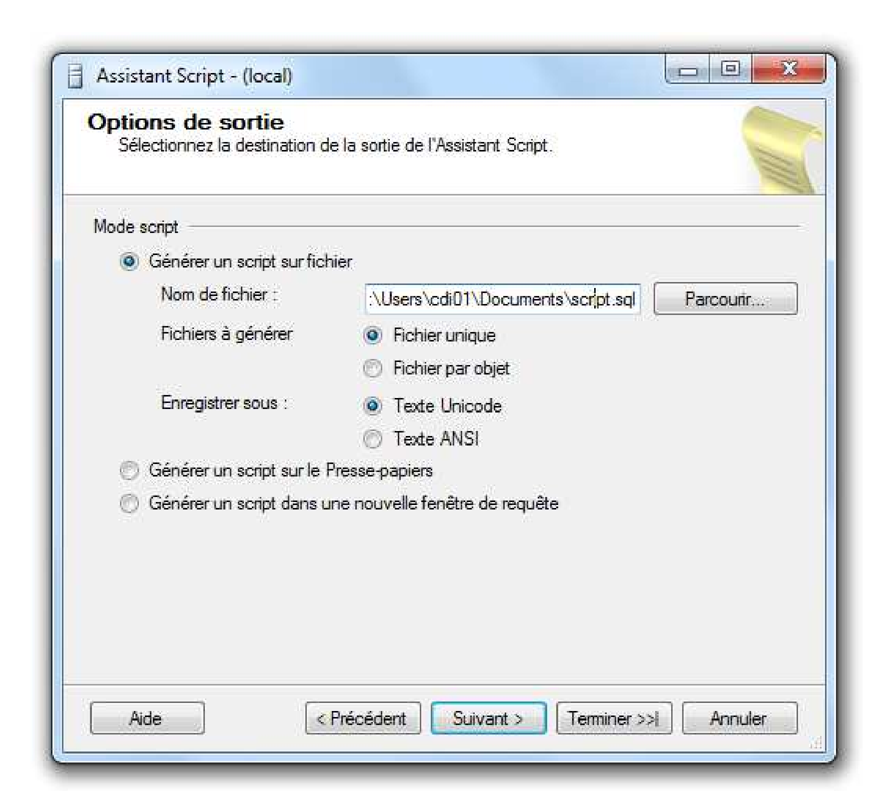


Puis toutes vos tables.

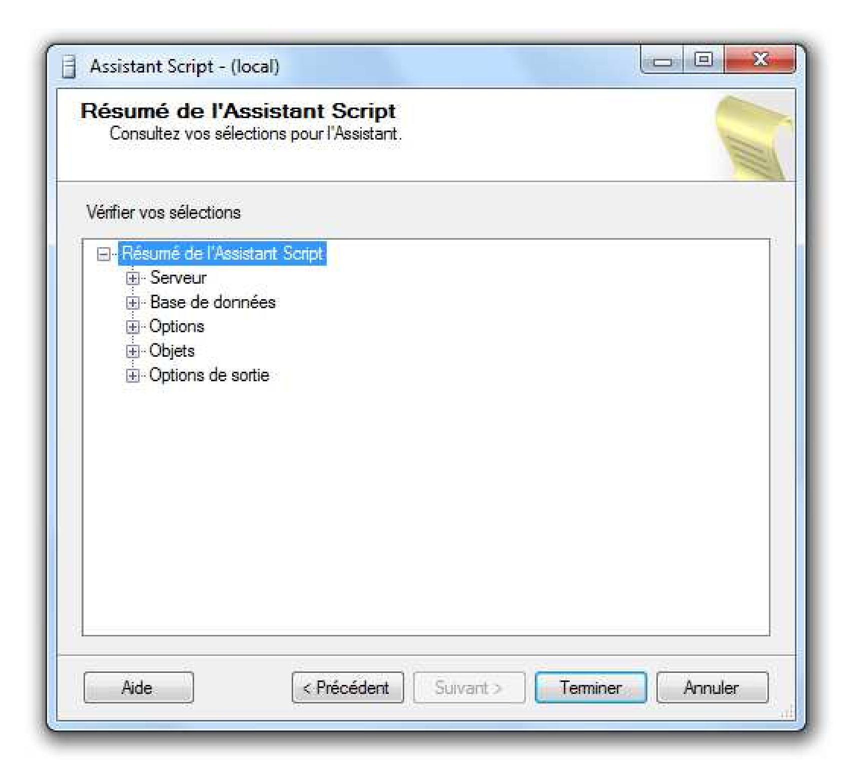
Puis cliquez sur « **Terminer** ».



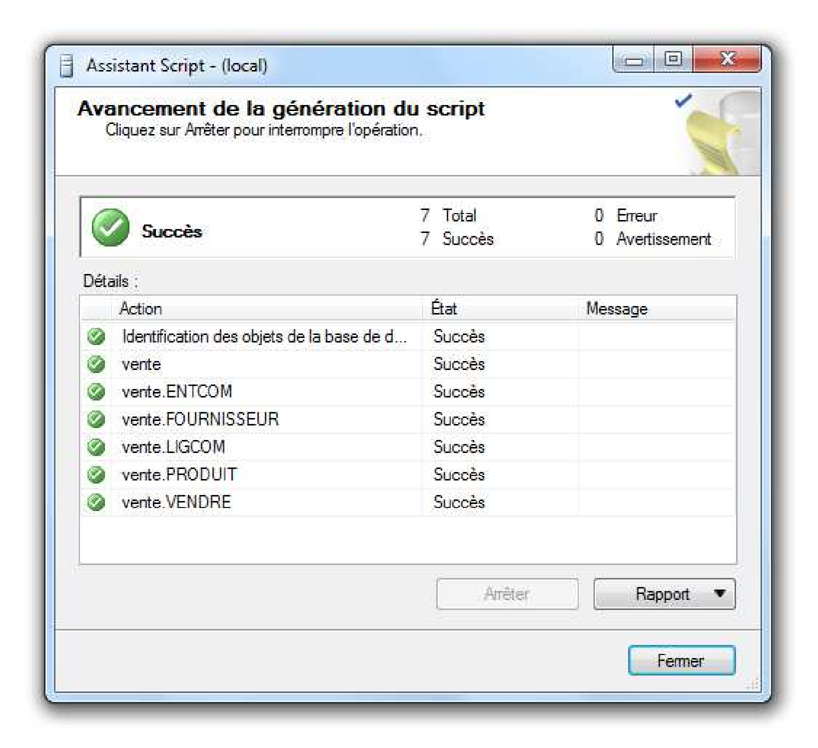
Configurez les options (nom du fichier et répertoire de sortie).



Puis cliquez sur « **Terminer** ».



Vous aurez alors un fichier *.sql dans le répertoire que vous avez indiqué.



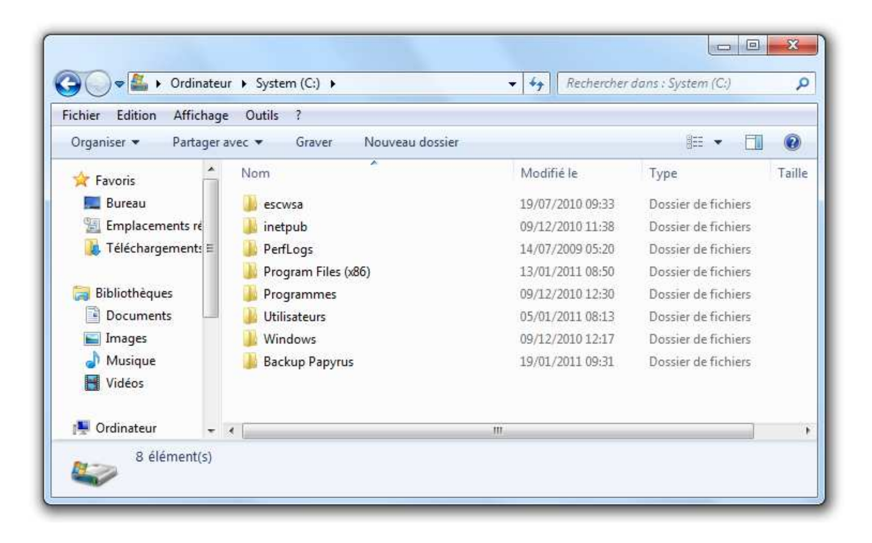
Sauvegarde de la base de données

Les bases de données utilisateurs sont les bases les plus sujettes à être sauvegardées dans l'entreprise. Il est important que dans SQL Server la sauvegarde d'une base de données ne se fasse pas sous forme de fichier, mais bien sous forme d'unité. On peut alors énoncer les unités physiques et les unités logiques de sauvegarde. Définissons ces deux termes :

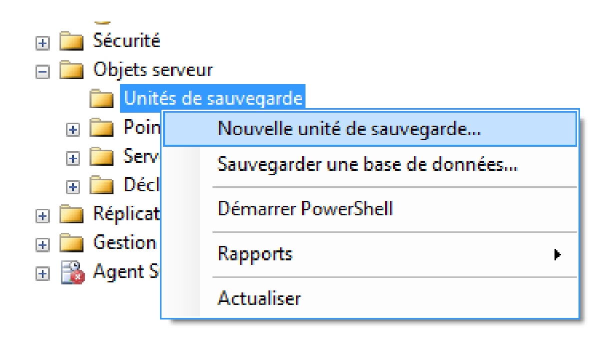
Unité de sauvegarde physique : Une unité de sauvegarde physique correspond au nom complet du fichier de sauvegarde dans le système de fichier Windows. Pour prendre un exemple parlant, si jamais une opération qui peut entraîner une perte de données est à faire, il convient d'effectuer une sauvegarde sur une unité physique de données, autrement dit, un disque.

Unité de sauvegarde logique : Une unité logique de sauvegarde est en vérité, une unité de sauvegarde physique référencée par un nom logique dans SQL Server.

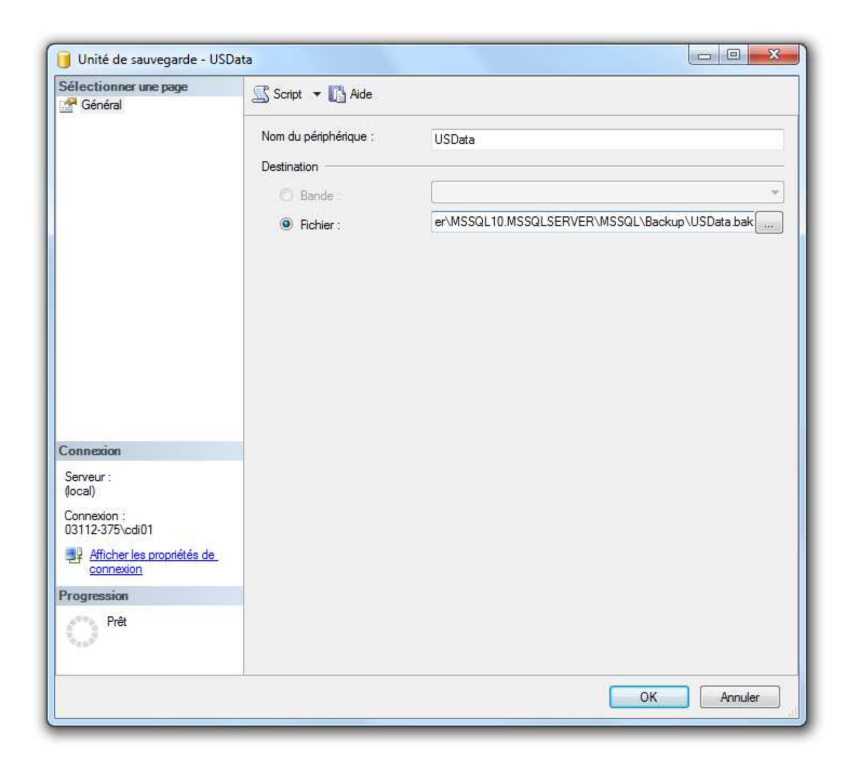
Créer un répertoire « Backup Papyrus » dans votre espace de travail. Nous créons le dossier sous « **C:/** » (ou répertoire de votre choix).



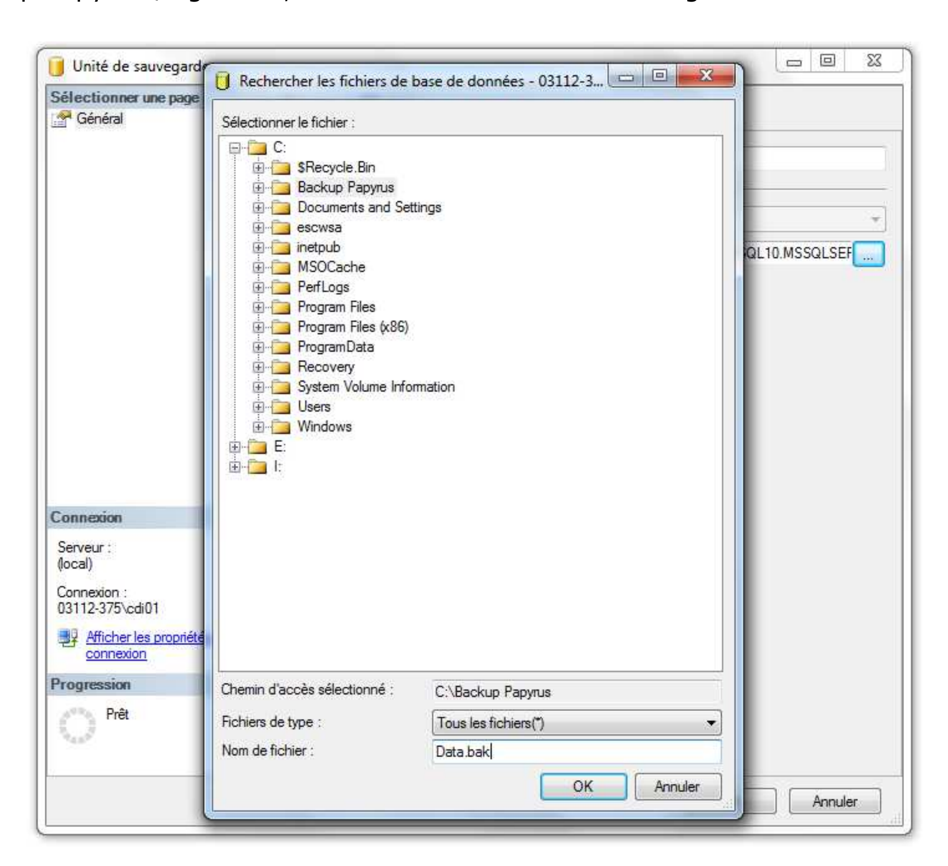
Sous Management Studio, onglet « **Objets Serveur** » :



Créez deux nouvelles unités de sauvegarde (USData pour les données, USLog pour les journaux).



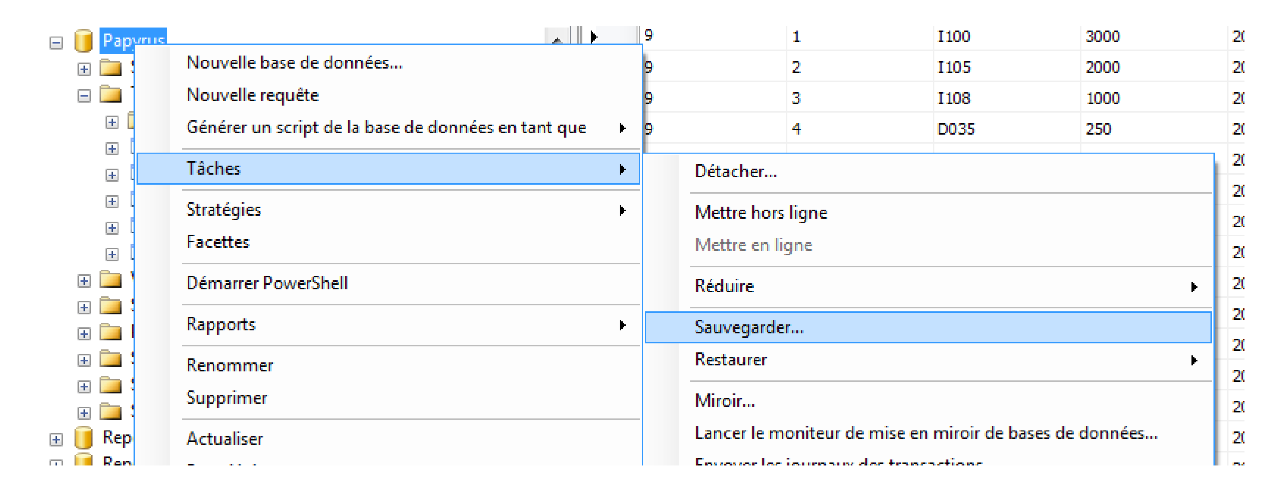
Vous associerez respectivement aux emplacements physiques « \Backup Papyrus\Data.bak » et « \Backup Papyrus\Log.bak », destinées à recevoir les sauvegardes de votre base Papyrus.



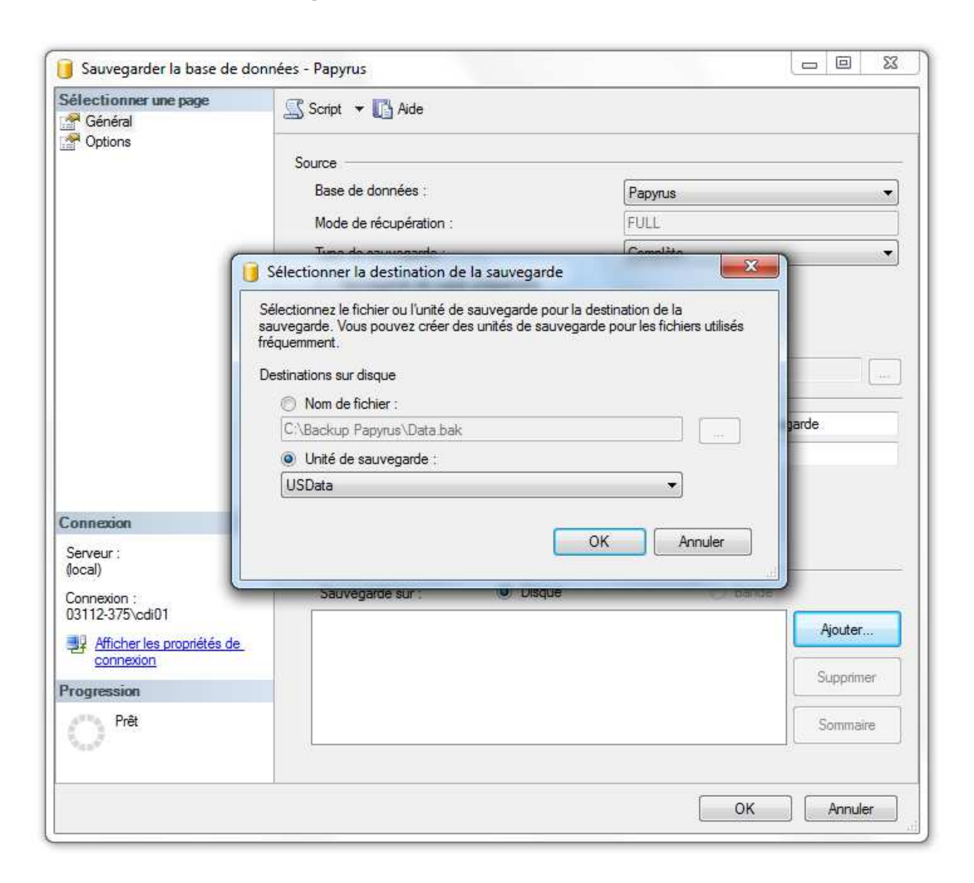
USData se trouvera donc dans le répertoire « C:\Backup Papyrus » avec le nom de fichier « Data.bak » et **USLog** se trouvera donc dans le répertoire « C:\Backup Papyrus » avec le nom de fichier « Log.bak ».

Sauvegarder par l'interface

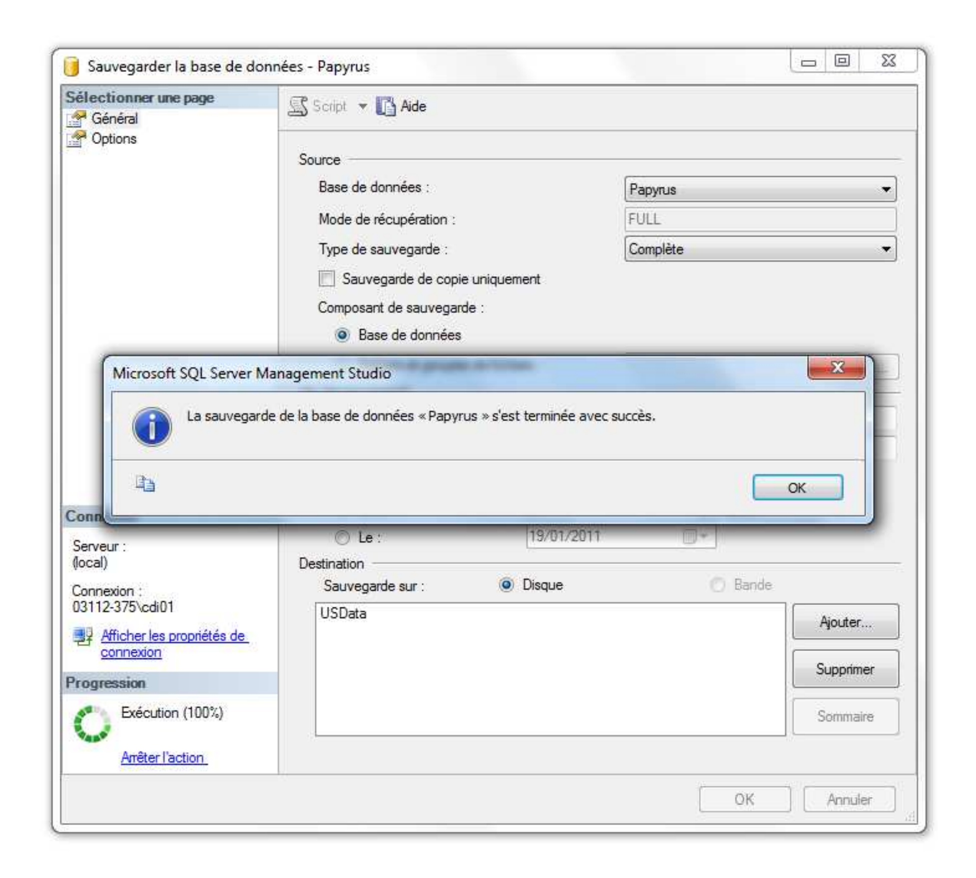
Cliquez droit sur la base de données « Papyrus » puis « **Tâches** » et « **Sauvegarder…** ».



Au lieu d'utiliser une destination sur le disque à partir d'un nom de fichier, on sélectionne directement notre unité de sauvegarde « **USData** ».



Puis cliquez sur « **OK** » pour lancer la sauvegarde.



Sauvegarder par le code

Le code suivant créer une sauvegarde complète de la base « Papyrus » à un emplacement que nous lui indiquons. On sauvegarde la base de données sous « Data.bak » et les logs sous « Log.bak ».

```
/**** Sauvegarde complète de la base ****/
BACKUP DATABASE [Papyrus] TO DISK = 'C:\Backup Papyrus\Data.bak'
WITH NOFORMAT, NOINIT,  NAME = 'Data.bak', SKIP, NOREWIND, NOUNLOAD,  STATS
= 10
GO
BACKUP LOG [Papyrus] TO DISK = 'C:\Backup Papyrus\Log.bak'
WITH NOFORMAT, NOINIT,  NAME = 'Log.bak', SKIP, NOREWIND, NOUNLOAD,  STATS
= 10
GO
```

Le code suivant créer une sauvegarde complète de la base exactement comme précédemment sauf que nous formatons les éventuelles sauvegardes précédentes grâce à l'option **WITH FORMAT**.

```
/**** Sauvegarde complète de la base ****/
BACKUP DATABASE [Papyrus] TO DISK = 'C:\Backup Papyrus\Data.bak'
WITH FORMAT, NAME = 'Data.bak', SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO
BACKUP LOG [Papyrus] TO DISK = 'C:\Backup Papyrus\Log.bak'
WITH FORMAT, NAME = 'Log.bak', SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO
```

Puisque nous avons précédemment créé des unités de sauvegardes, le plus simple est alors de les utiliser. Dans ce cas, nous n'avons plus besoin d'indiquer le chemin du répertoire.

```
/**** Sauvegarde complète de la base ****/
BACKUP DATABASE [Papyrus] TO USDATA
WITH FORMAT, NAME = 'Data.bak', SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO
BACKUP LOG [Papyrus] TO USLOG
WITH FORMAT, NAME = 'Log.bak', SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO
```

Sauvegarde différentielle

Les sauvegardes de journaux peuvent vite devenir volumineuses, il existe une alternative saine pour pallier à ce problème, les sauvegardes différentielles de base de données. En effet, ce type de sauvegarde va nous permettre de sauvegarder les données non sauvegardées dans la dernière sauvegarde totale. Cette solution est donc évolutive, rapide et elle permet un gain de place énorme.

« Créez un script effectuant une sauvegarde différentielle de la base Papyrus »

```
/**** Sauvegarde Base avec description et différentielle ****/
BACKUP DATABASE [Papyrus] TO USLog
WITH description = 'Sauvegarde de la base différentielle',differential,
NOFORMAT, NOINIT,  NAME = 'JT1.bak', SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO
```

Avec **WITH FORMAT** on réinitialise le support

```
/**** Sauvegarde Base avec description et différentielle ****/
BACKUP DATABASE [Papyrus] TO USLog
WITH description = 'Sauvegarde de la base différentielle',differential,
FORMAT, NAME = 'JT1.bak', SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO
```

Ou en indiquant le répertoire de destination

```
/**** Sauvegarde Base avec description et différentielle ****/
BACKUP DATABASE [Papyrus] TO DISK = 'C:\Backup Papyrus\JT1.bak'
WITH description = 'Sauvegarde de la base différentielle',differential,
FORMAT, NAME = 'JT1.bak', SKIP, NOREWIND, NOUNLOAD,  STATS = 10
GO
```

Sauvegarde par groupes de fichiers

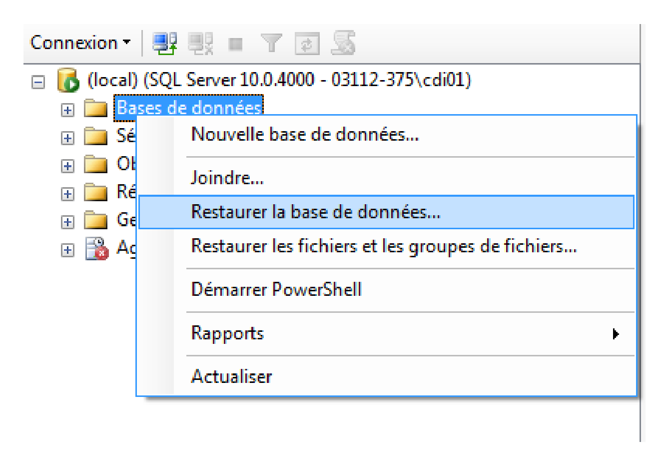
Les sauvegardes complètes et les sauvegardes différentielles de base de données peuvent prendre s'étendre sur la durée lorsqu'il s'agit de bases de données à gros volumes. Si toutefois votre base de données est définie sur plusieurs fichiers de données (.mdf), il est possible de choisir de sauvegarder la base par groupe de fichiers de données.

Restauration de la base de données

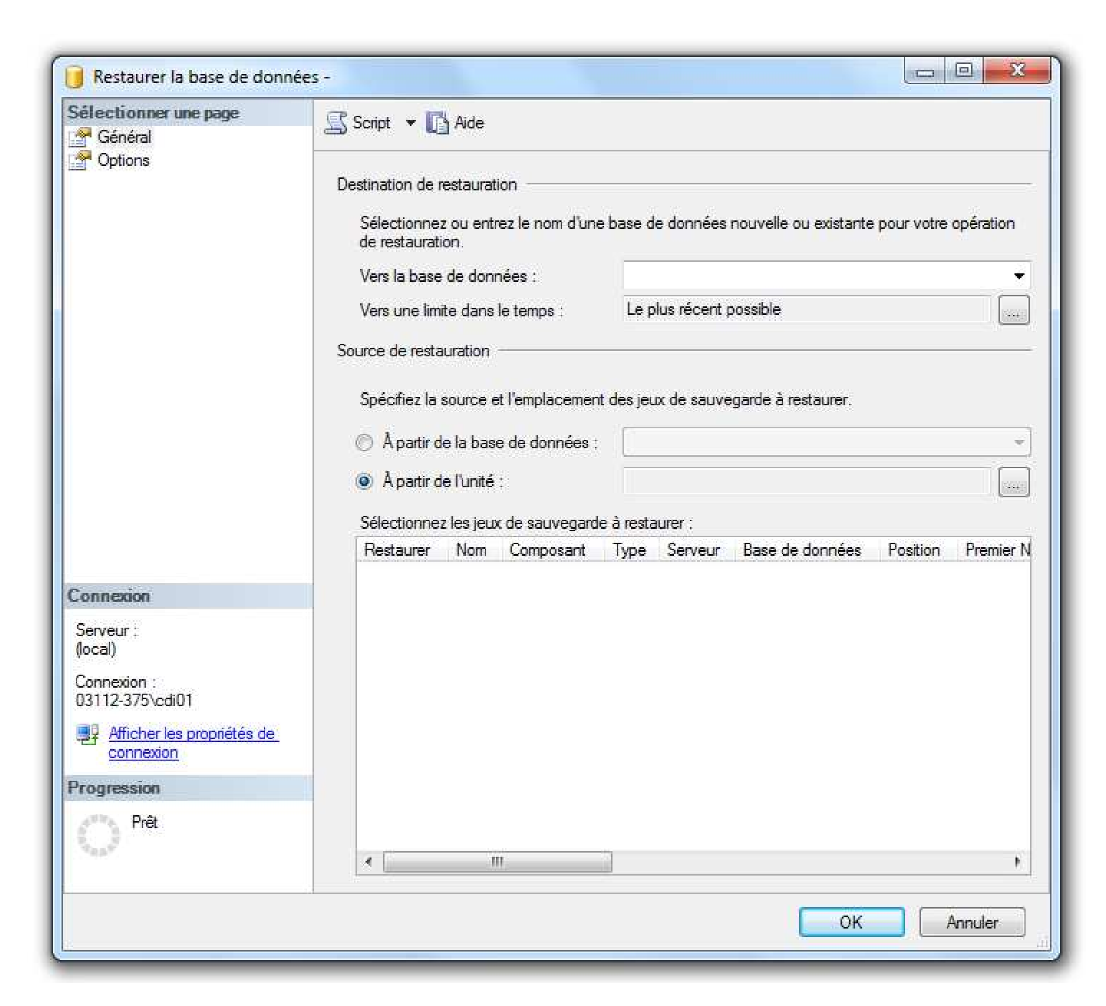
Une opération de restauration de base de données correspond à l'opération inverse de la sauvegarde de base de données. Celle-ci peut être faite de deux manières : grâce à l'instruction **RESTORE** ou par l'intermédiaire de l'interface graphique dans SSMS.

Restauration par l'interface

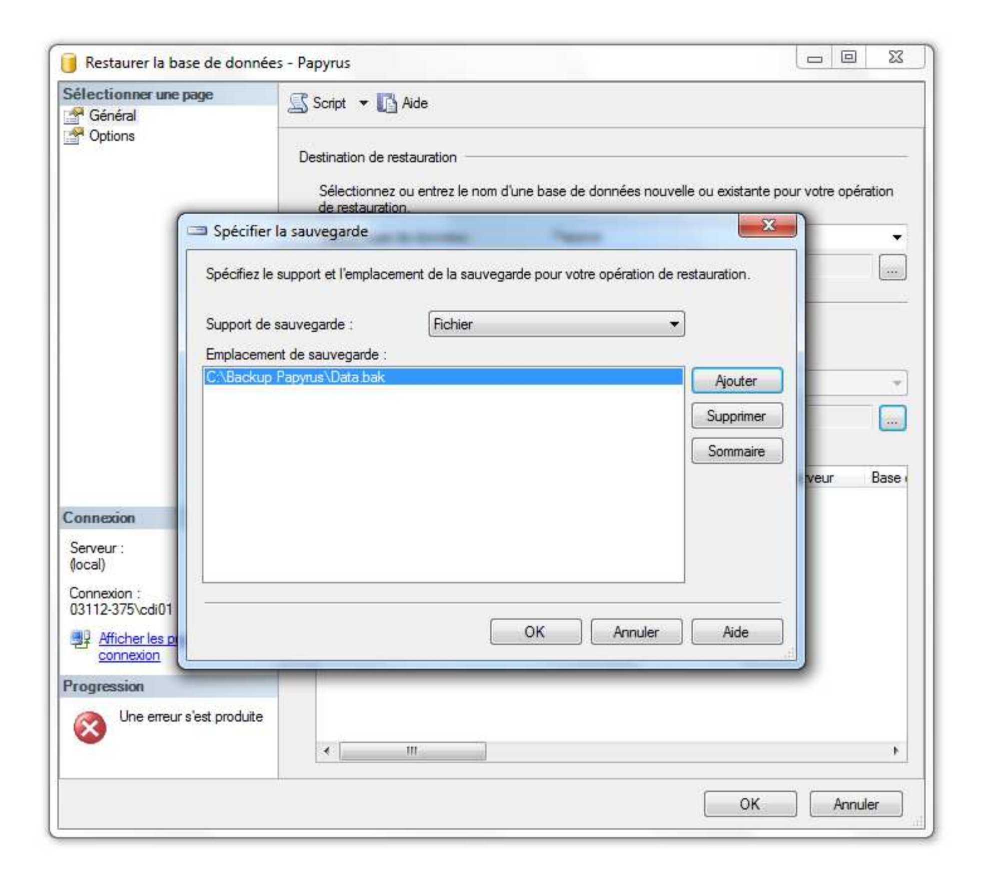
Sur le dossier « **Bases de données** », cliquez droit et « **Restaurer la base de données…** ».



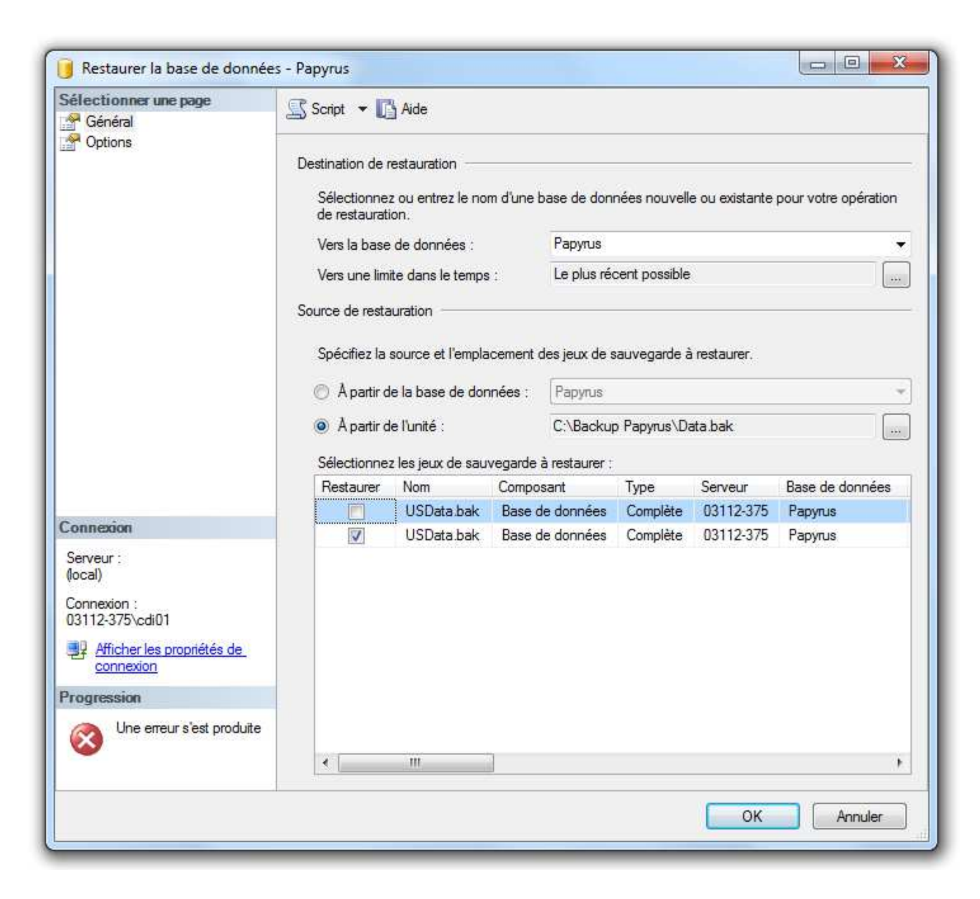
Vous avec alors plusieurs options. Si on restaure à partir de l'unité comme dans le choix suivant :



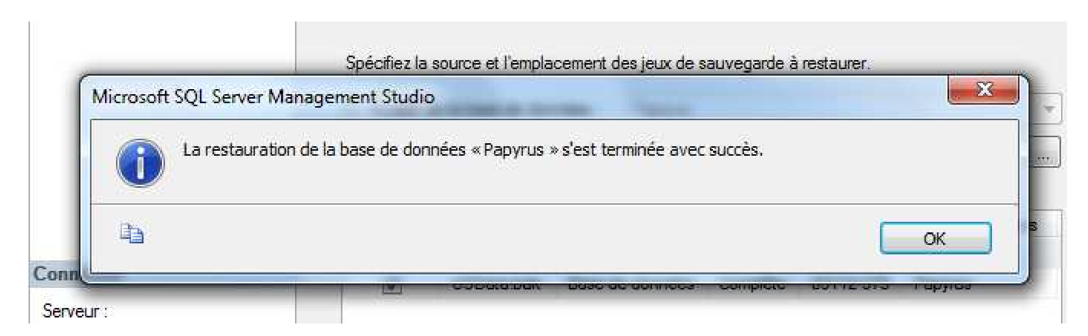
On spécifie le support et l'emplacement du fichier « **Data.bak** » pour notre exemple.



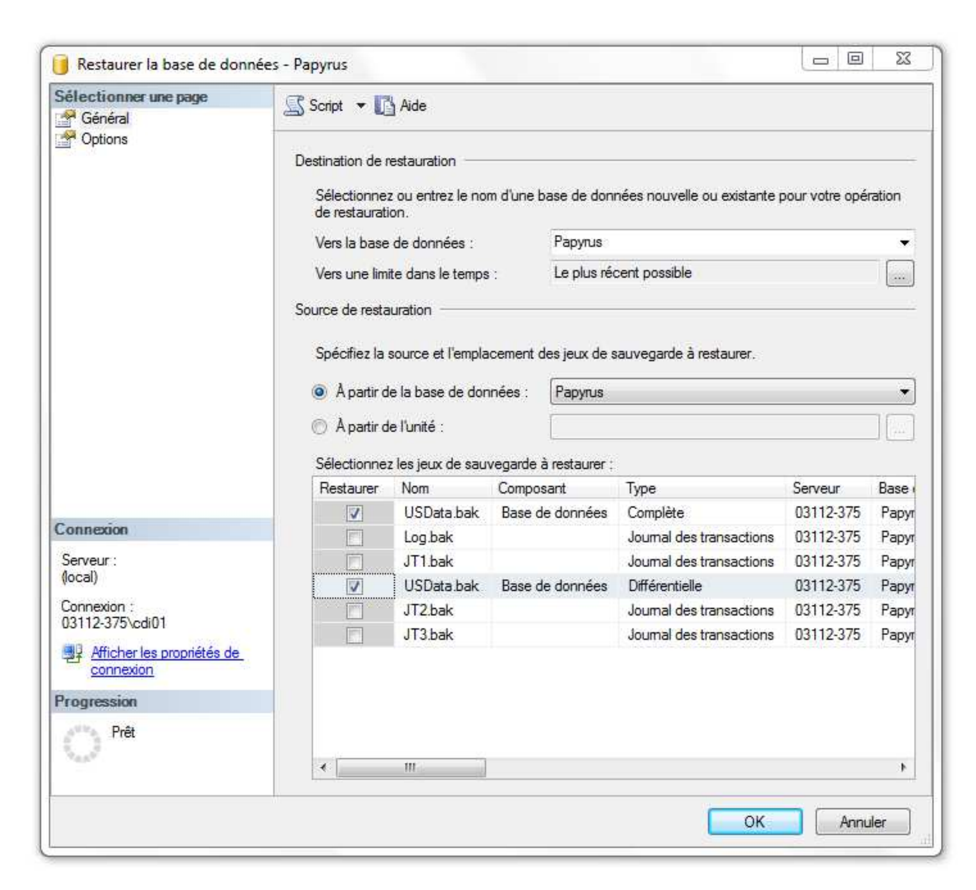
Sélectionnez alors le jeu de sauvegarde à restaurer puis cliquez sur « **Ok** ».



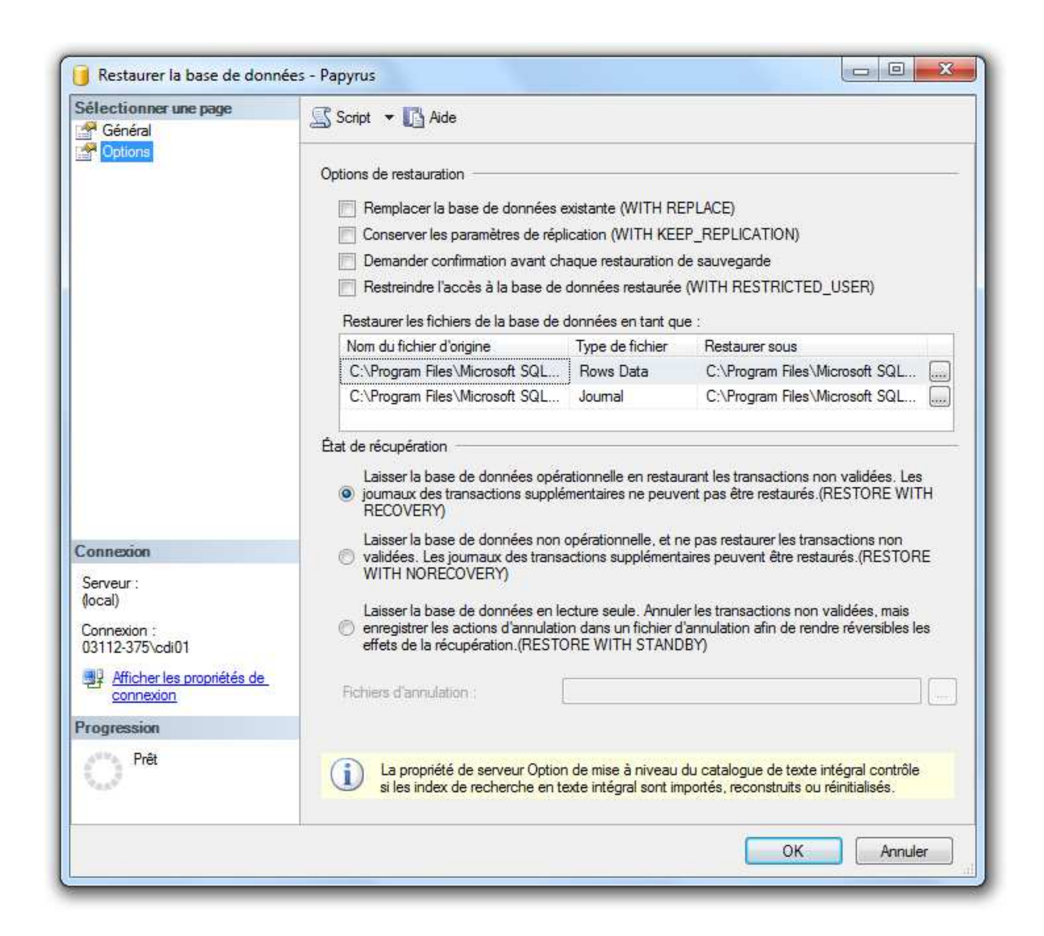
Un message de confirmation permet de savoir si la base de données a été restaurée dans son intégralité.



Vous pouvez ne restaurer qu'une partir des données… Par exemple, si on avait choisi les options suivantes :



Vous avez alors accès à différentes options :



Vous pouvez alors autoriser l'accès à la base à tous les utilisateurs (accès que vous aviez restreint au démarrage de l'opération de restauration). Pour reverrouiller la base, il vous faudrait utiliser le code suivant :

```
/**** Verrouillage de la base ****/
ALTER DATABASE papyrus SET RESTRICTED_USER WITH ROLLBACK IMMEDIATE
```

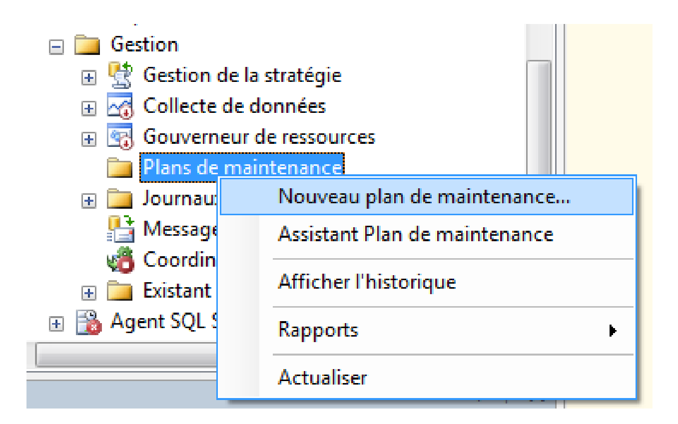
Restauration par le code

Avec du code T-SQL, c'est l'instruction **RESTORE** qui va nous permettre d'effectuer une restauration de base de données :

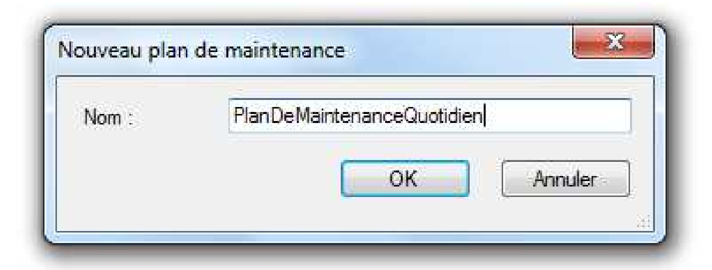
```
/**** Restauration de la base après suppression ****/
RESTORE DATABASE [Papyrus] FROM  DISK = N'C:\Backup Papyrus\Data.bak' WITH
FILE = 1,  NORECOVERY,  NOUNLOAD,  STATS = 10
RESTORE DATABASE [Papyrus] FROM  DISK = N'C:\Backup Papyrus\Data.bak' WITH
FILE = 2,  NOUNLOAD,  STATS = 10
GO
```

Plan de maintenance

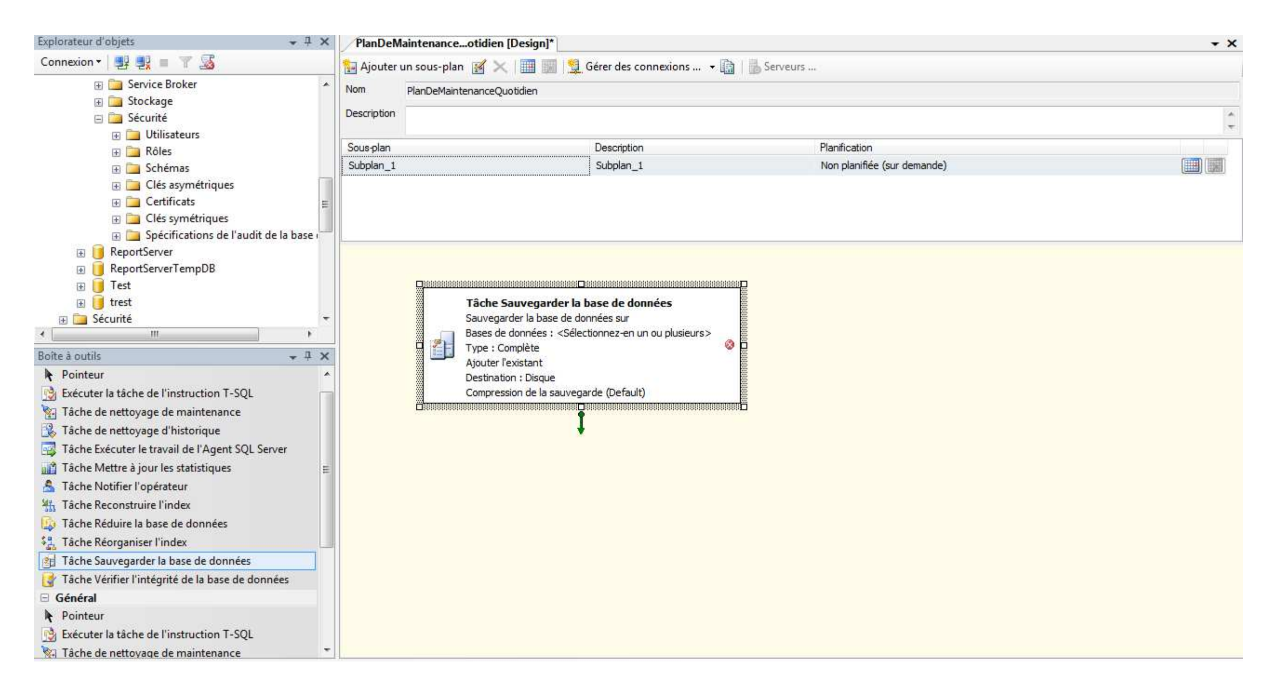
Vous pouvez créer plusieurs plans de maintenance pour vos bases de données. Ouvrez le dossier « **Gestion** » de « Microsoft SQL Server Management Studio » puis cliquez droit au niveau du dossier « **Plans de maintenance** » puis « **Nouveau plan de maintenance…** ».



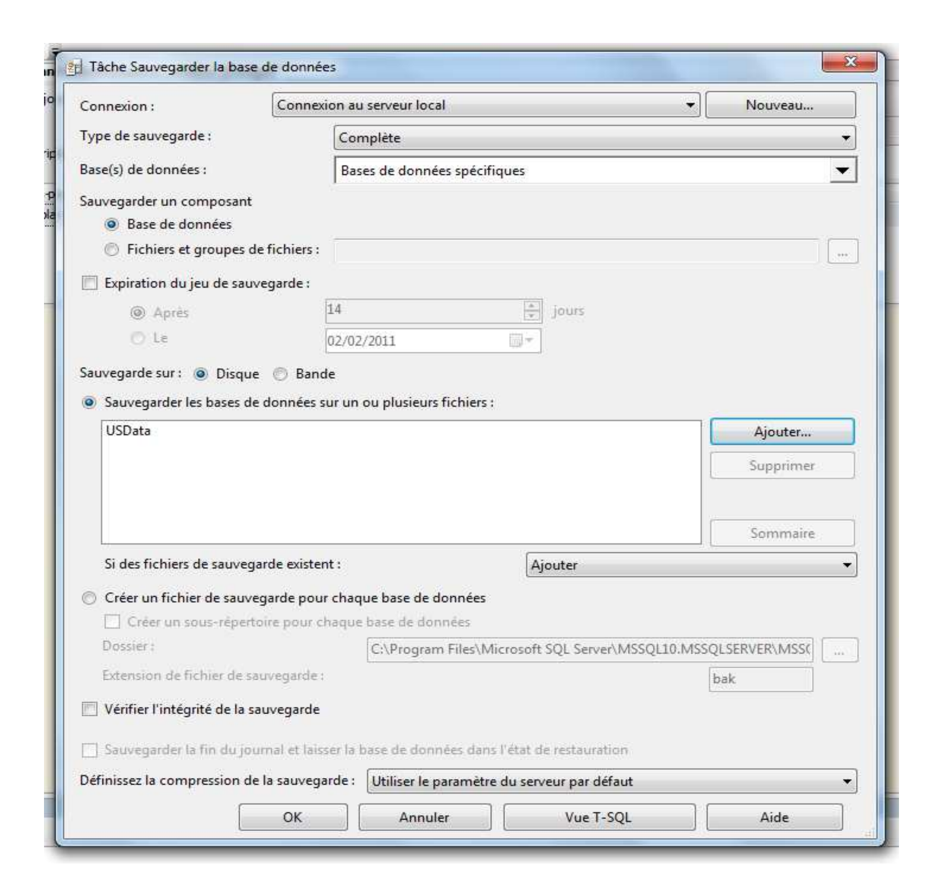
Donner un nom à votre plan de maintenance.



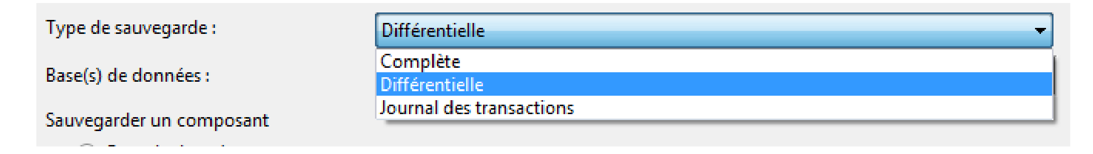
Utilisez la « **Boîte à outils** » à gauche de l'écran et utilisez la souris pour cliquez et faire glisser l'action de maintenant souhaitée. Ici, on fait glisser la « **Tâche Sauvegarder la base de données** ».



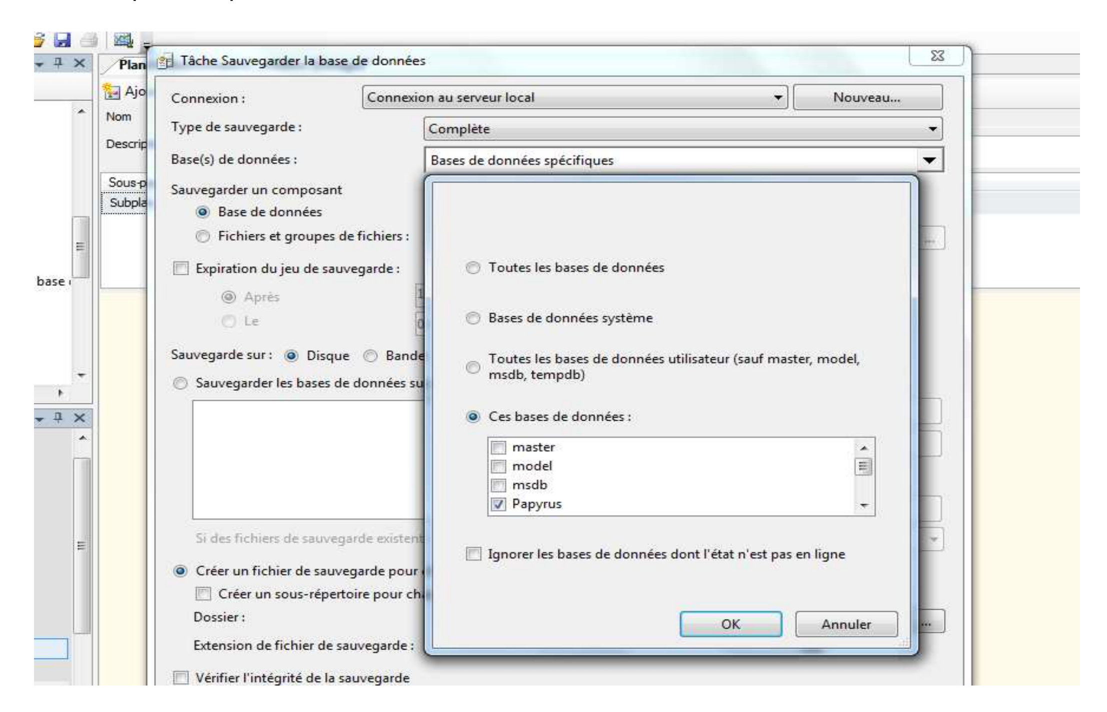
Double cliquez sur la tâche que vous venez d'ajouter pour la paramètré.



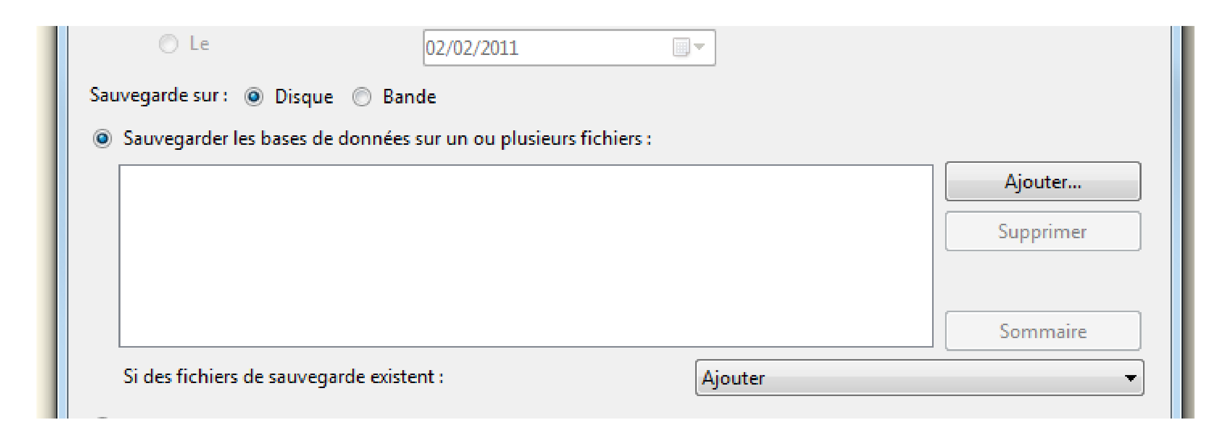
Sélectionnez le type de sauvegarde.



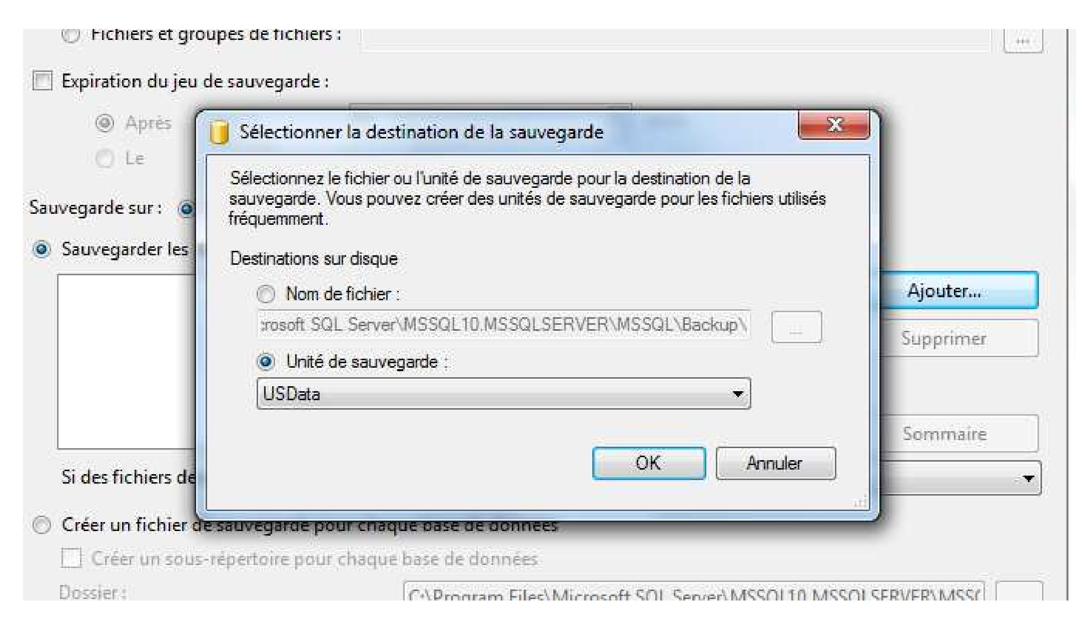
Sélectionnez la base de données (ici « Papyrus ») concernée par votre plan de maintenance en la cochant puis cliquez sur « **Ok** ».



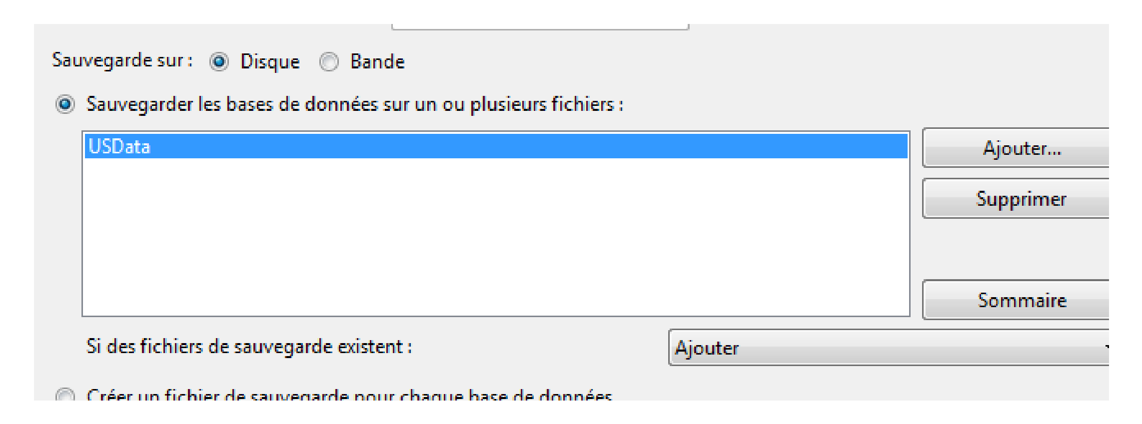
Cochez « **Sauvegarder les bases de données sur un ou plusieurs fichiers** » et cliquez sur « **Ajoutez** ».



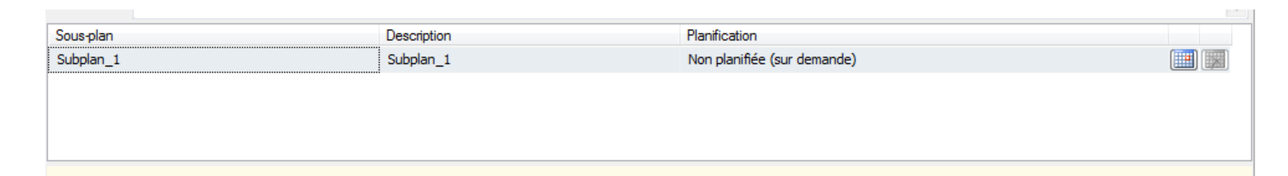
Sur la boîte de dialogue qui s'affiche, on sélectionnera notre unité de sauvegarde (où éventuellement le nom de fichier où se trouve notre sauvegarde).



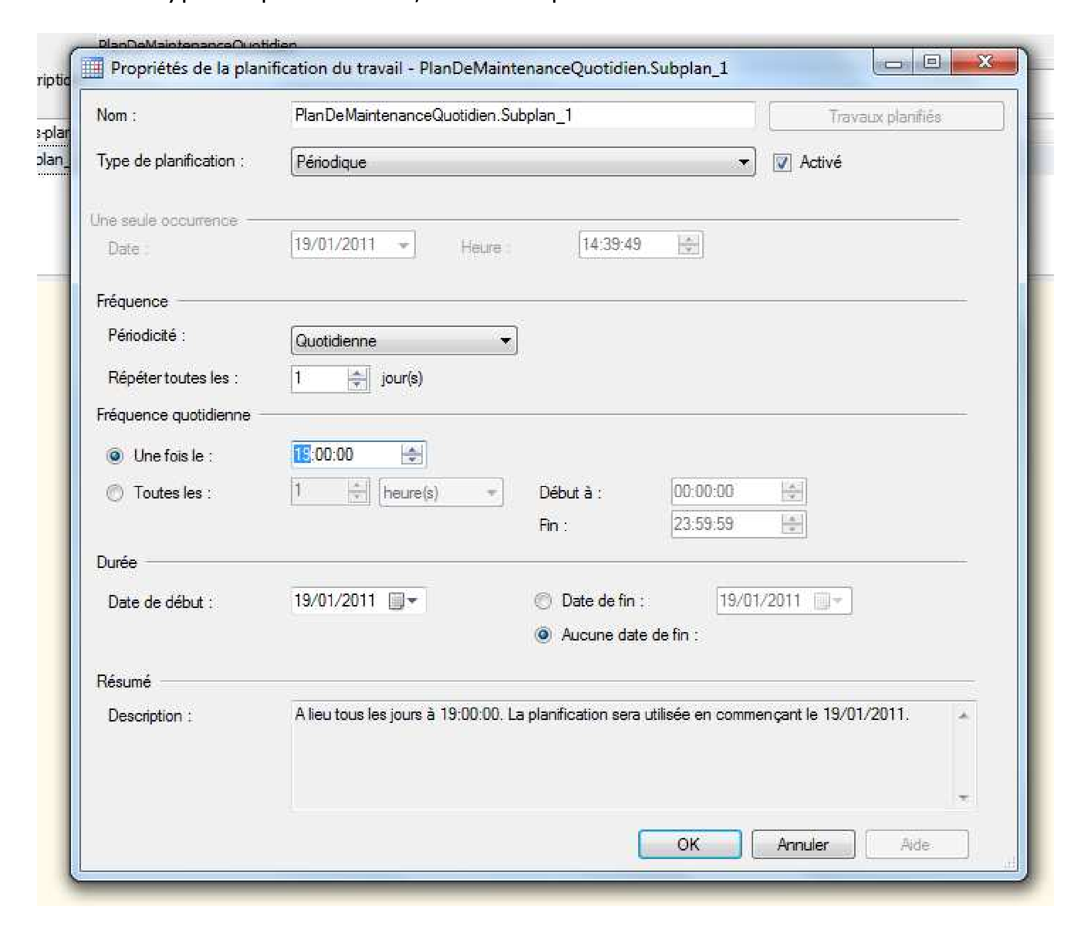
USData s'affiche alors. Cliquez sur « **Ok** » en bas de la fenêtre.



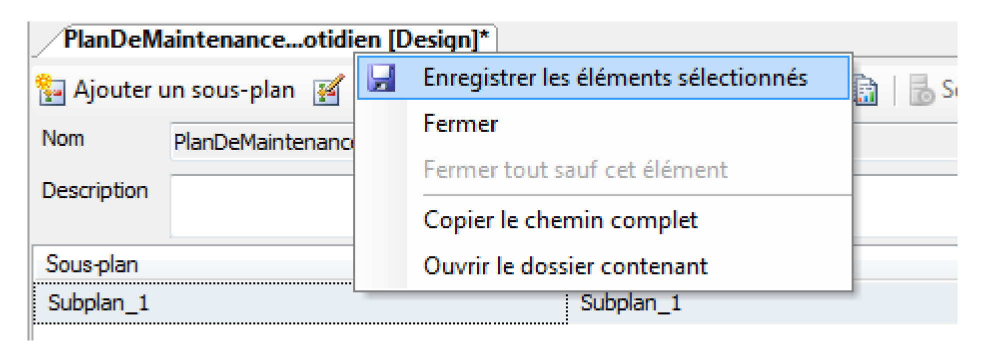
Cliquez sur le calendrier du sous-plan afin de configurer la périodicité.



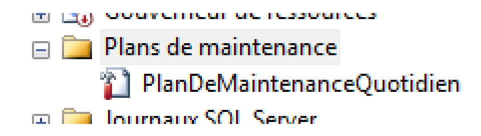
Choisissez votre type de planification, votre fréquence…



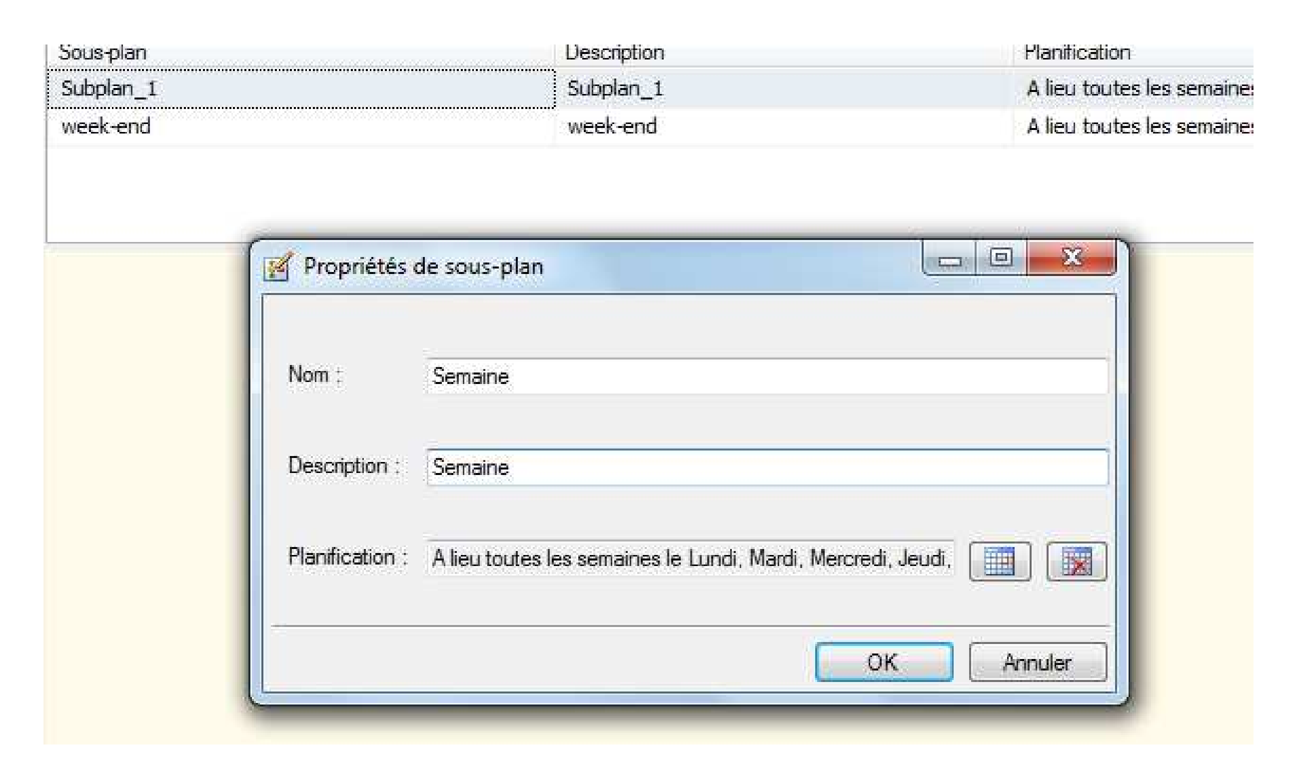
Vous pouvez ajouter plusieurs sous-plans de maintenant, les renommés à votre guise et choisir la planification de votre choix. N'oubliez pas de le sauvegarder.



Celui-ci apparaît alors dans le dossier « **Plans de maintenance** » de l'interface.



Pour renommer un sous-plan, cliquez deux fois dessus.



Plusieurs sous-plans permettent par exemple d'avoir un sous-plan pour la sauvegarde des journaux, les jours ouvrés toutes les 2 heures et un autre sous plan pour la sauvegarde des journaux le week-end toutes les 4 heures.

Gestion des accès serveur

Avant de pouvoir commencer à travailler avec les données de nos bases, il est impératif de se logger sur le serveur SQL. Cette étape permet de s'identifier au niveau du serveur SQL, afin de pouvoir exploiter les droits qui ont étés attribués à notre connexion. Dans SQL Server, il existe plusieurs modes d'authentification que nous allons définir dans cette partie.

Mode de sécurité Windows : Dans ce mode de sécurité, SQL Server s'appuie sur les fonctionnalités de Windows. En effet, il se sert de la sécurité de Windows (login et mot de passe) pour créer des connexions aux instances et donc aux bases de données. Puisque le mode de sécurité Windows se sert de la sécurité du système sur lequel SQL Server est installé, la gestion des groupes est aussi possible. Il est donc plus facile d'administrer des connexions aux bases de données, de manière plus souple qu'utilisateur par utilisateur.

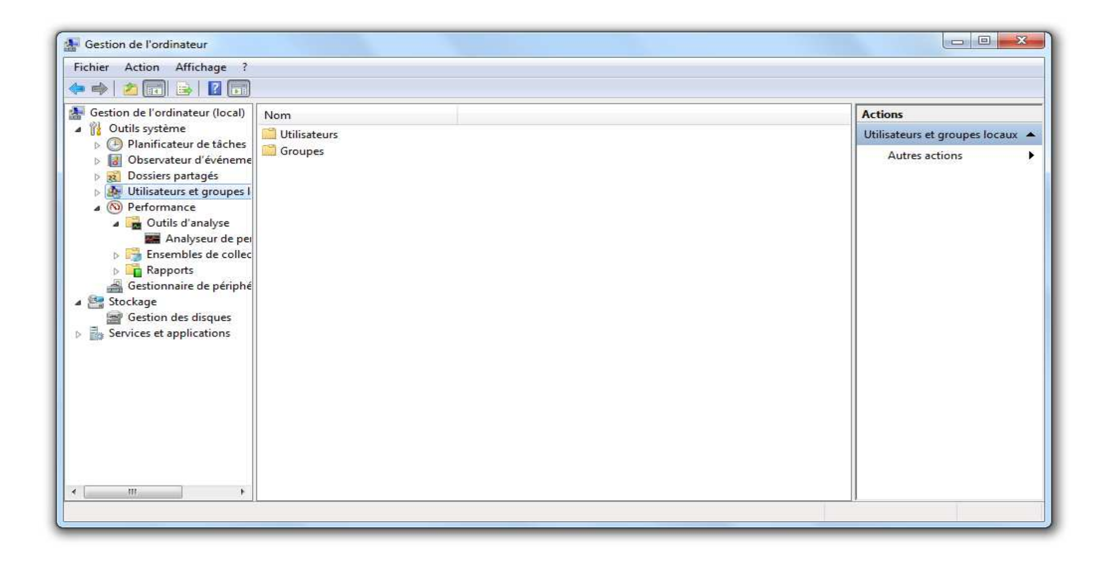
Mode de sécurité Mixte : Le mode de sécurité mixte repose sur une double authentification : l'authentification Windows puis l'authentification SQL Server. C'est le mode le plus connu pour la sécurité des bases de données. Dans le principe, c'est SQL Server qui se charge de vérifier que l'utilisateur existe et qu'il possède le bon mot de passe. Cela signifie que les utilisateurs sont entièrement gérés par SQL Server, autant les logins que les mots de passe. Ce type d'identification est bien adapté pour une gestion des utilisateurs qui ne passent pas par une authentification Windows à la base.

Gestion des connexions à SQL Server

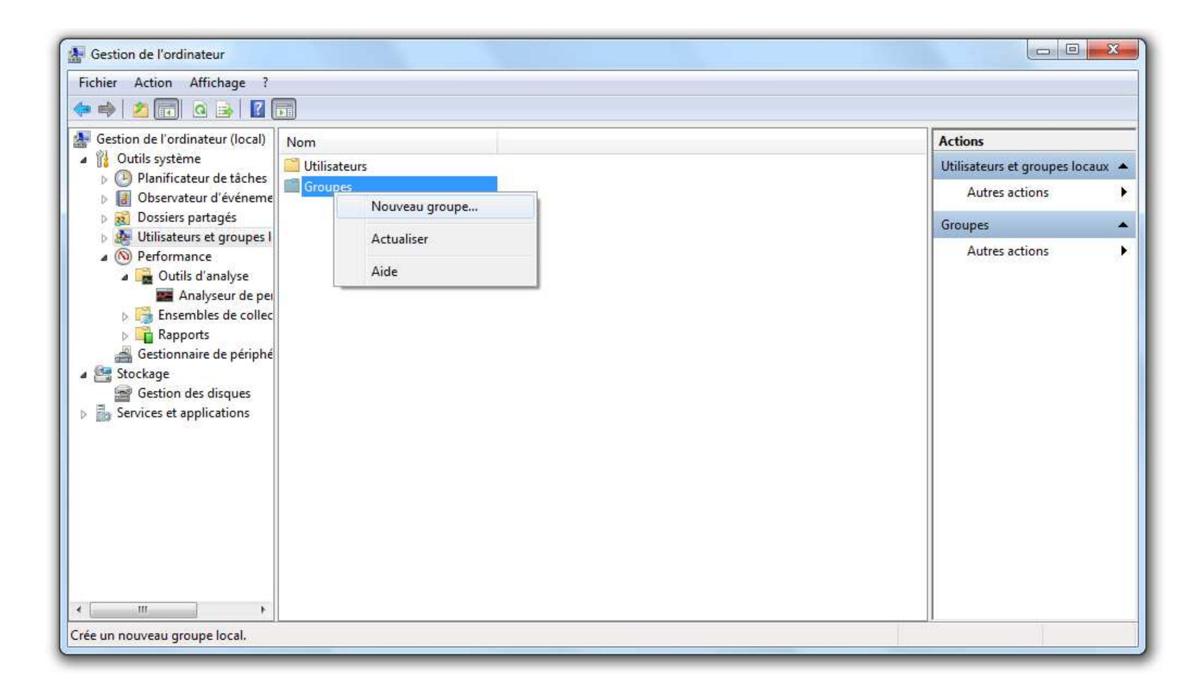
Créer les profils de connexion sous Windows

En mode sécurité Windows, les noms de groupes ou les utilisateurs doivent être les mêmes que sous Windows. Il existe plusieurs façons de créer des connexions au serveur de base de données. Nous allons les détailler. Mais avant, voyons comment créer les comptes de connexion sous Windows :

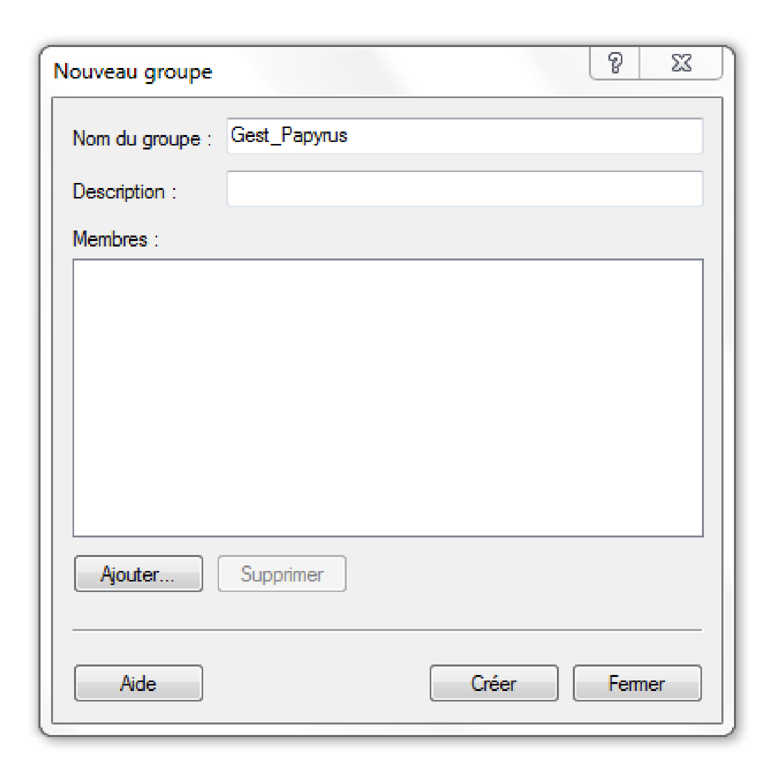
Dans Windows, allez dans le « **Panneau de configuration** » puis faire « **Outils d'administration** », « **gestion de l'ordinateur** » et « **Utilisateurs et groupes locaux** ».



Cliquez droit sur l'onglet des groupes puis « **Nouveau groupe…** »



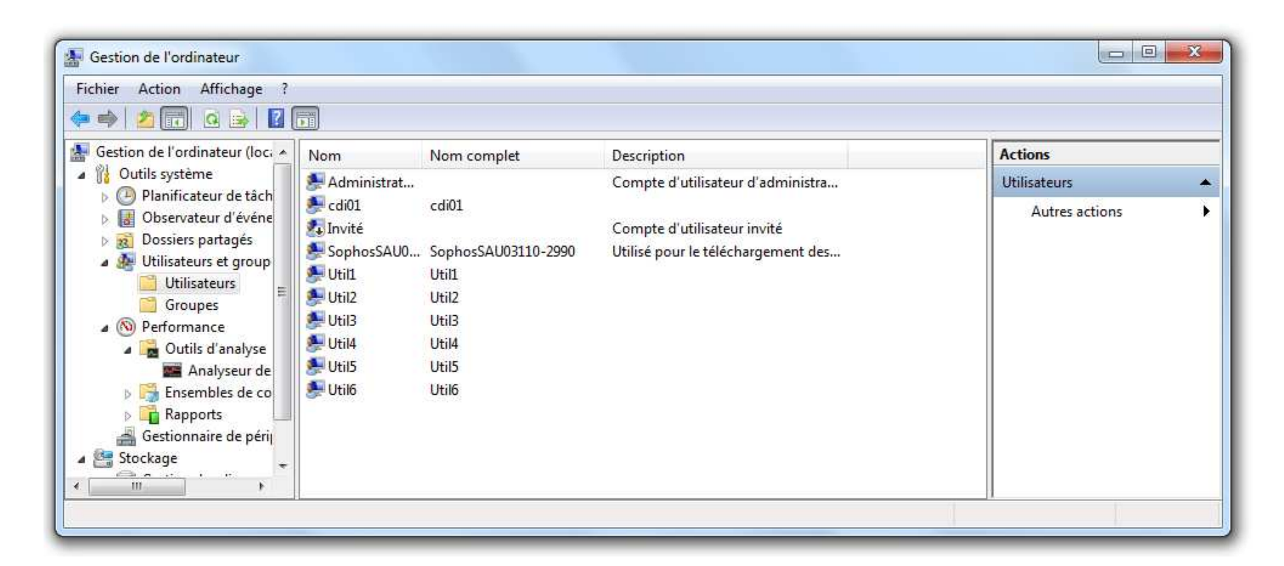
Donner un nom au groupe que vous souhaitez créer puis cliquer sur Créer. Le groupe créé apparaît alors dans le dossier des groupes.



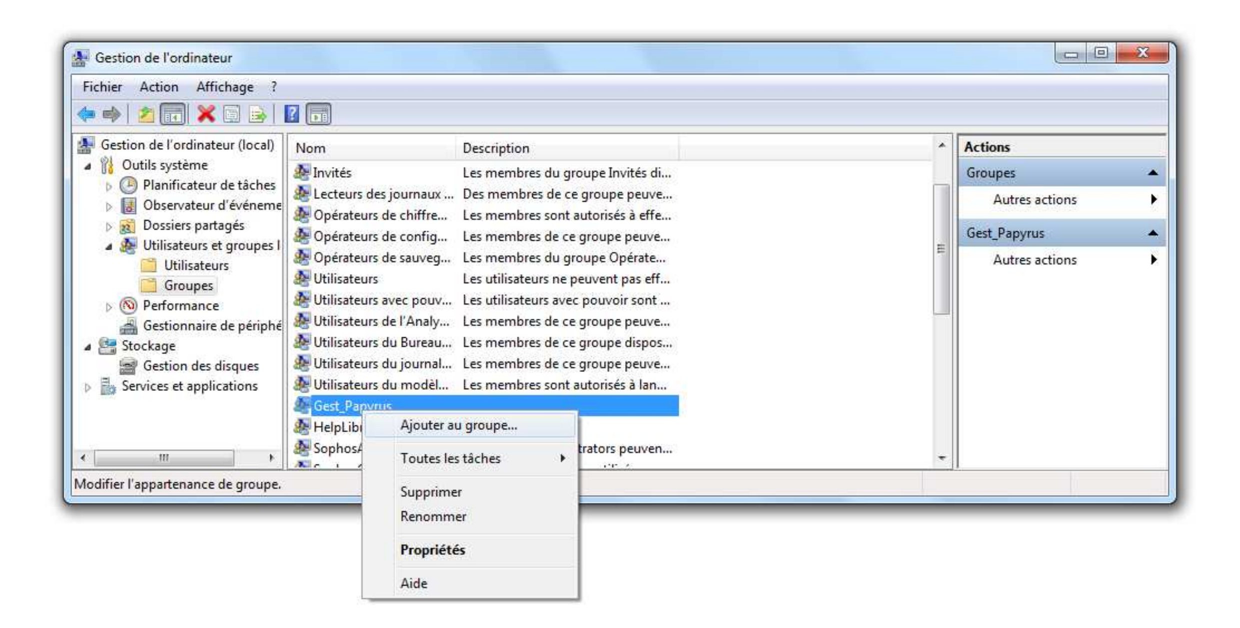
Pour créer un nouvel utilisateur, cliquer droit sur le dossier « **Utilisateurs** » puis donner lui un nom, un mot de passe et configurer les options de votre choix.



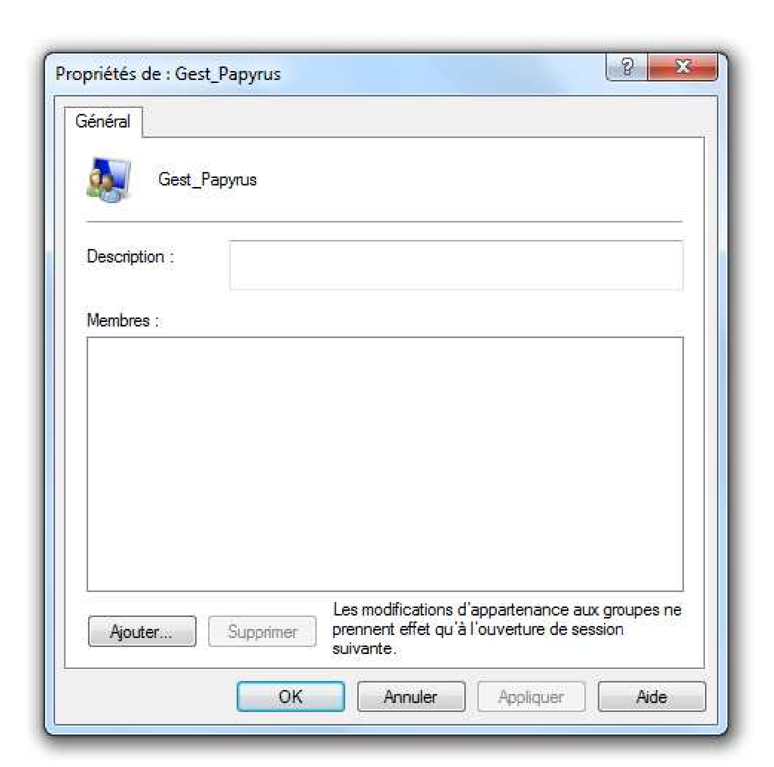
L'utilisateur créé apparaît alors dans le dossier des utilisateurs.



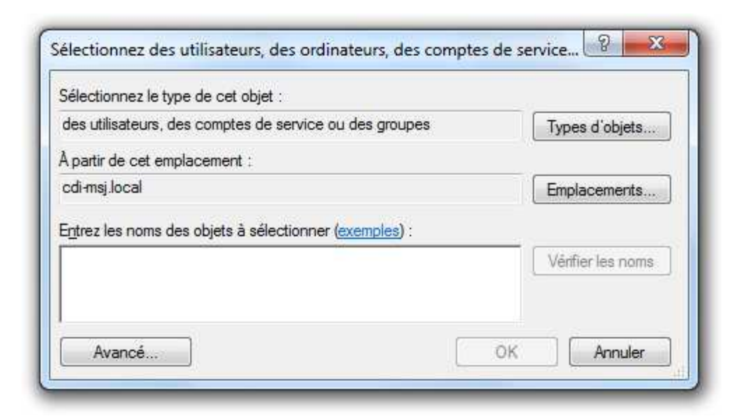
Pour ajouter un ou plusieurs utilisateurs à un groupe, allez dans le dossier des groupes, sélectionnez le groupe concernez puis cliquez droit dessus et « **Ajouter au groupe…** ».



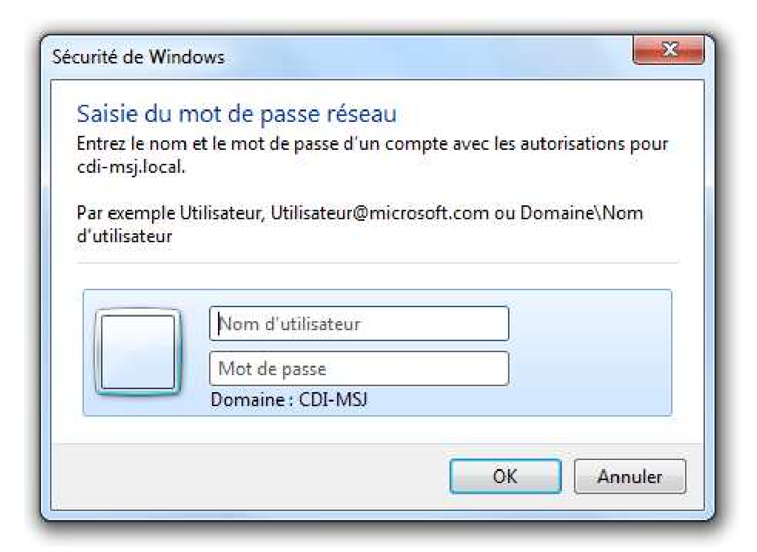
Sur la fenêtre qui apparaît, cliquez sur « **Ajouter…** ».



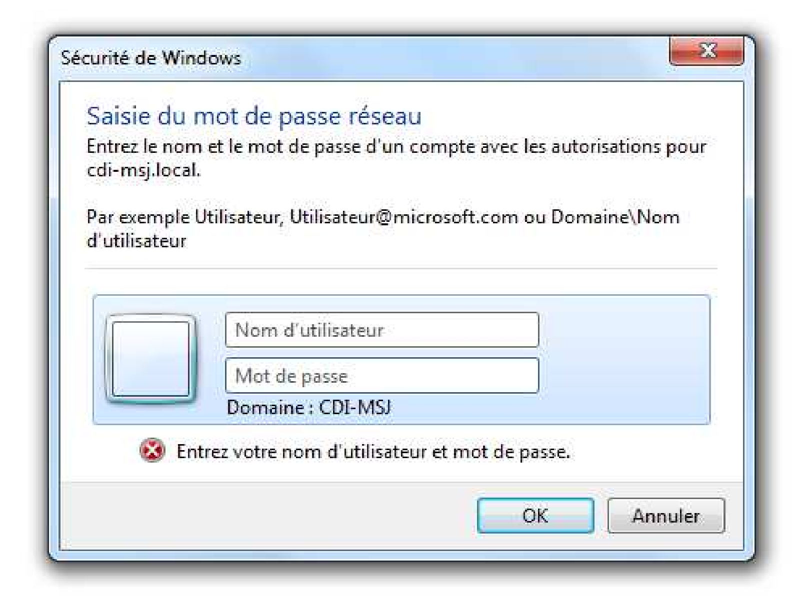
Modifions l'emplacement, car nous travaillons sur le domaine « 03112-375 » dans notre exemple. Cliquez donc sur « **Emplacements…** ».



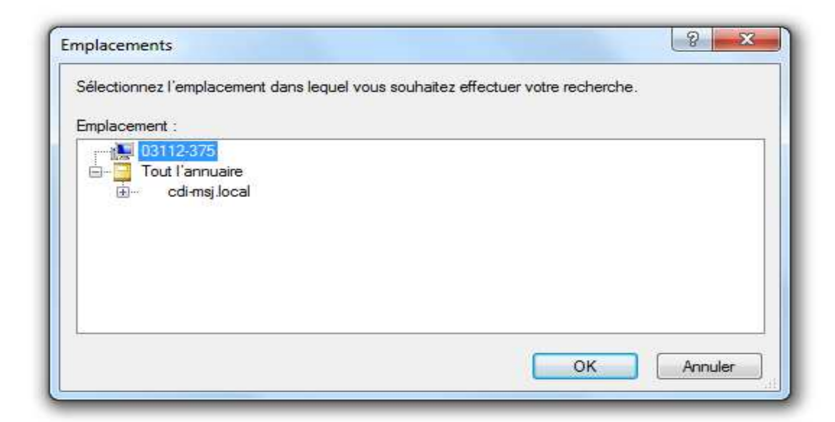
Ne vous préoccupez pas de la fenêtre suivante et cliquez sur « **OK** ».



Cliquez sur « **Annuler** ».



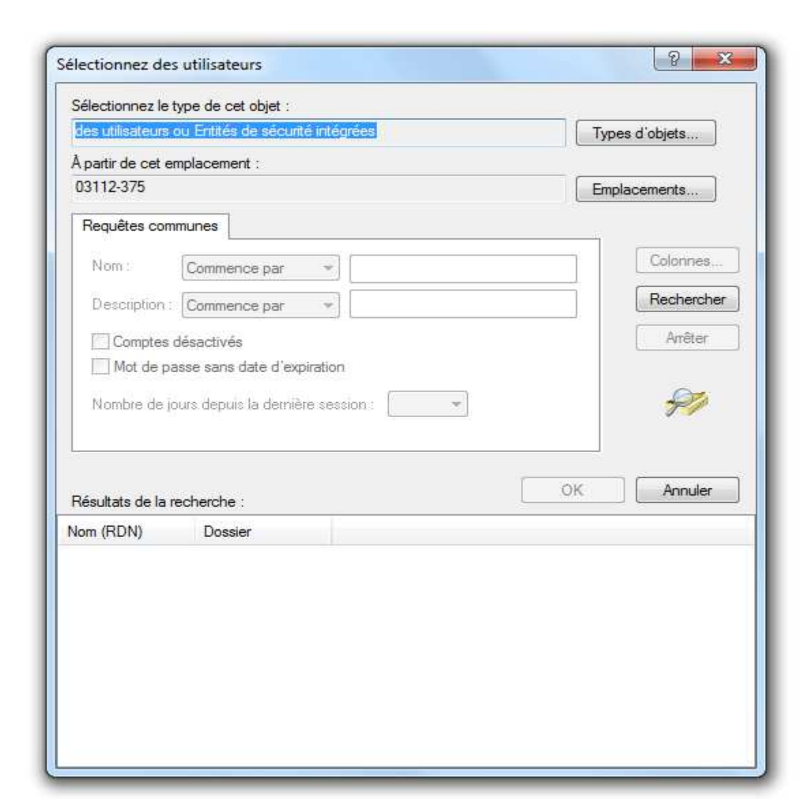
Sélectionnez le domaine concerné (ici « 03112-375 ») puis cliquez sur « **Ok** ».



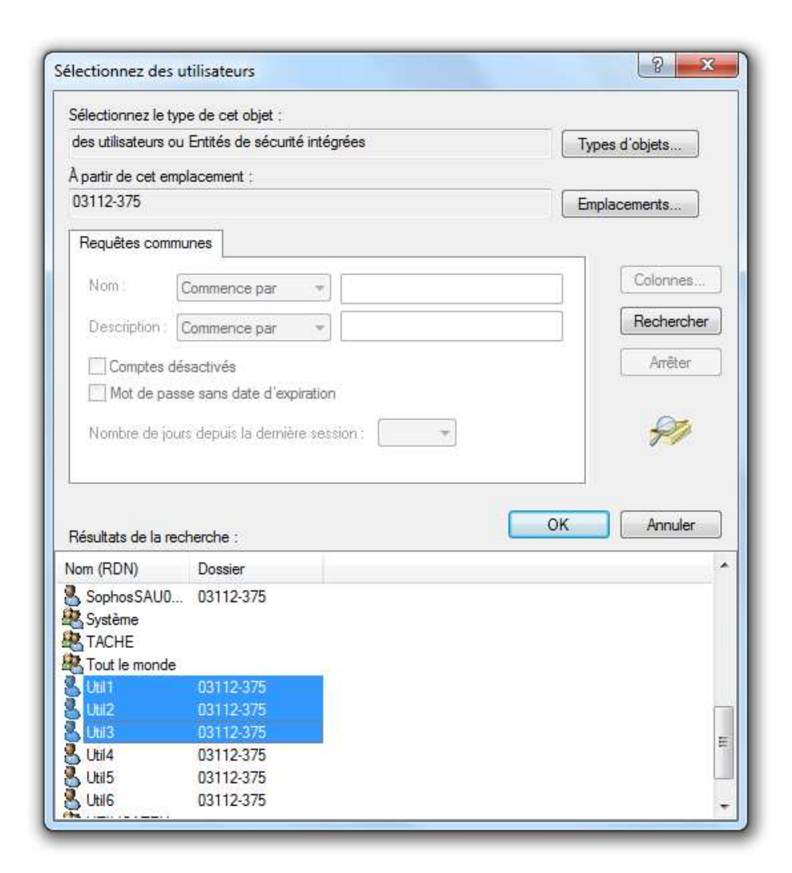
Vous constatez que l'emplacement a changé. Cliquez maintenant sur « **Avancée** ».



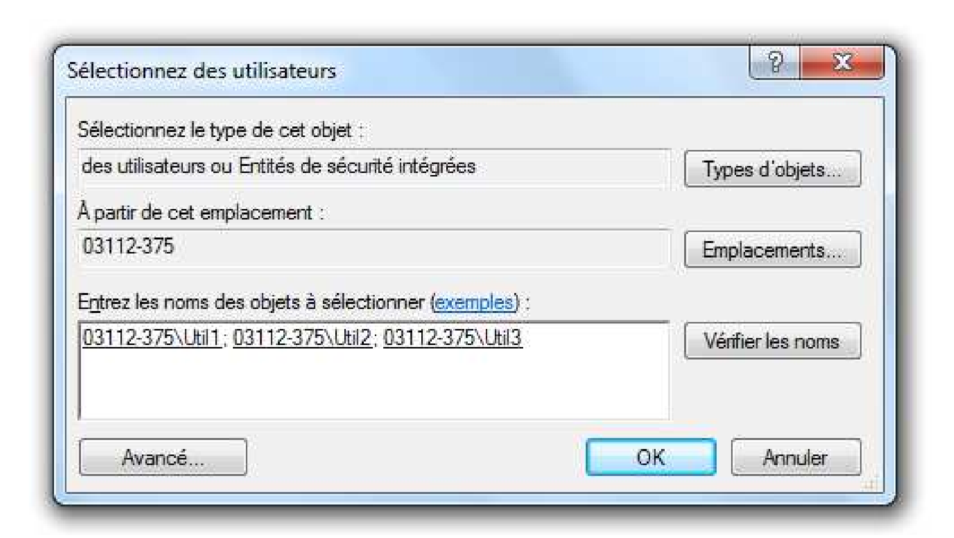
Cliquez alors sur « **Rechercher** ».



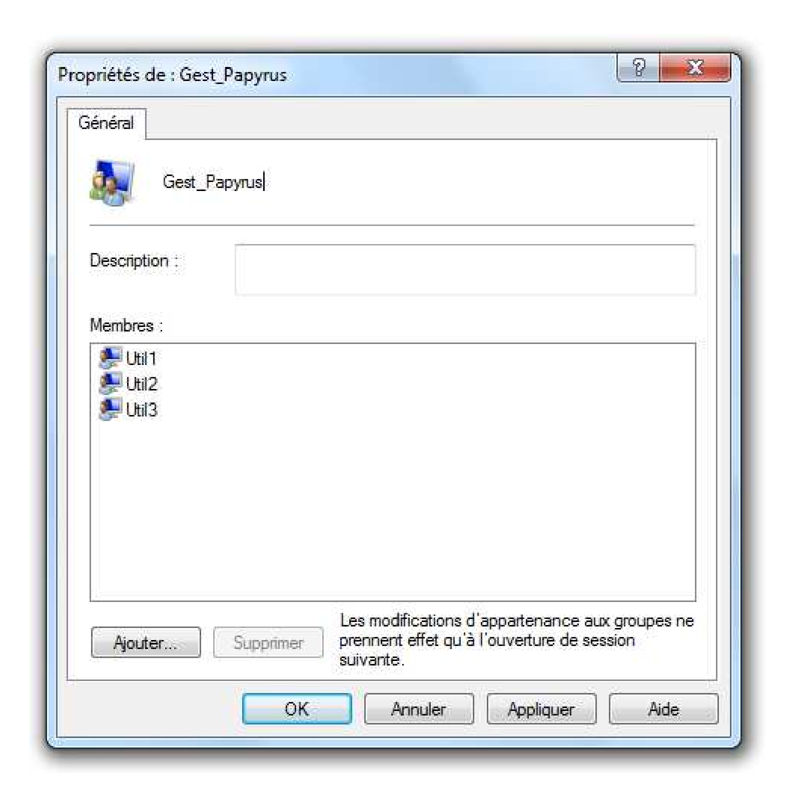
Vous pouvez maintenant sélectionner le ou les utilisateur(s) que vous souhaitez ajouter à votre groupe puis cliquez sur « **Ok** ».



Et de nouveau, cliquez sur « **Ok** ».



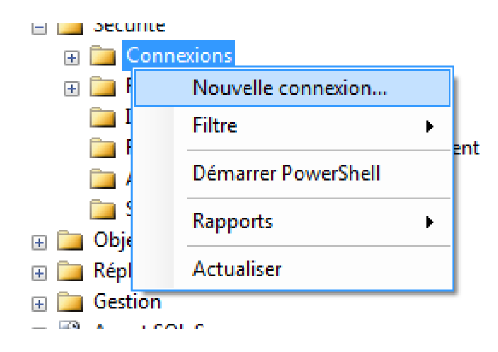
Vous venez d'ajouter les membres suivants (Util1, Util2 et Util3) au groupe nommé ici « Gest_Papyrus ».



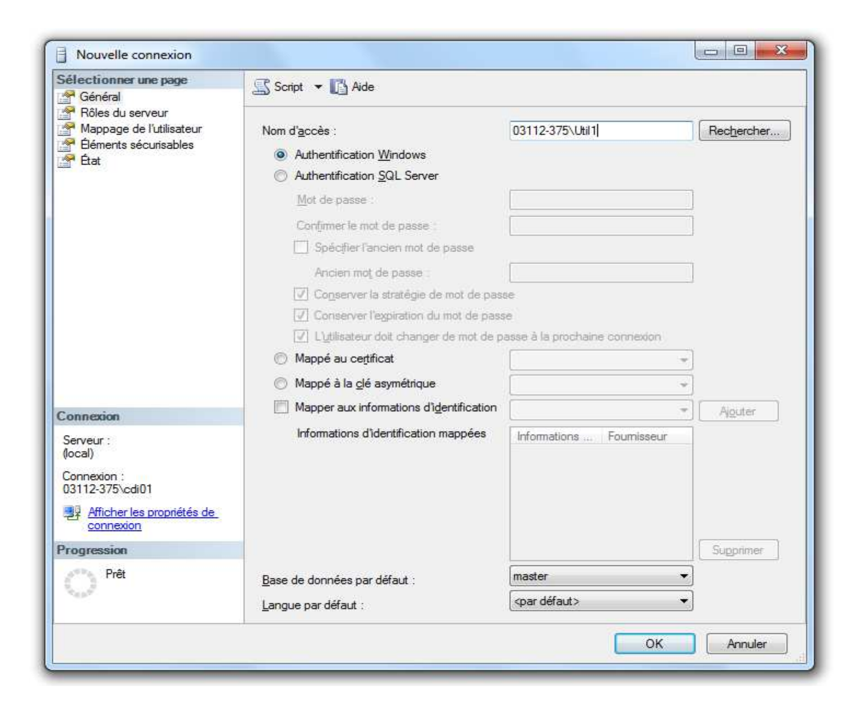
Créer les profils de connexion au serveur

En utilisant l'interface

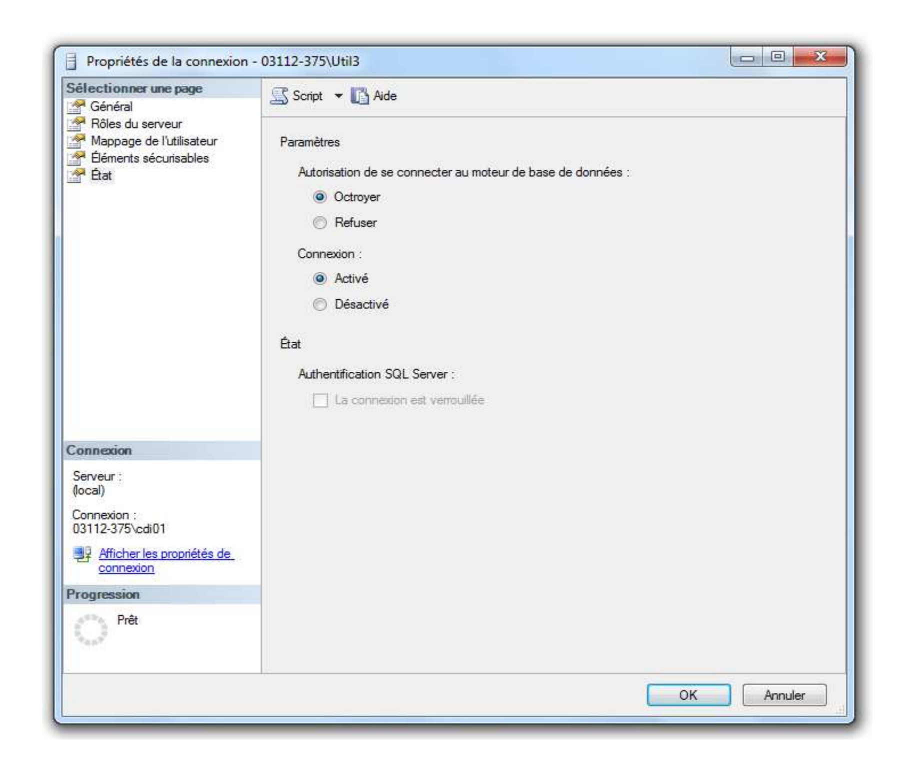
Cliquez droit sur le dossier « **Connexions** » et **« Nouvelle connexion… »**.



Donnée un nom d'accès (Nom de domaine + « \ » + Nom d'utilisateur) et définissez le type d'authentifications : Windows pour le mode sécurité Windows ou SQL Server pour le mode de sécurité mixte. Si vous choisissez s Authentification SQL Server, il vous restera donc plus qu'à donner un nom à la connexion et un mot de passe dans le champ prévu à cet effet. Cliquez sur « **OK** ».



L'interface graphique SSMS permet tout aussi bien de modifier les propriétés d'une connexion ou bien de la supprimer simplement. Sur l'onglet « **Etat** » :



Par le code

En mode sécurité Windows :

Avec du code T-SQL, la syntaxe de création d'une connexion SQL Server est la suivante : (pour rappel, les noms de groupes ou les utilisateurs doivent être les mêmes que sous Windows).

```
CREATE LOGIN nom_connexion
FROM WINDOWS
WITH DEFAULT_DATABASE=Master
DEFAULT_LANGUAGE=langue
```

L'instruction **CREATE LOGIN** permet d'annoncer à SQL Server que nous allons créer une connexion. Il est nécessaire de donner le nom de la connexion Windows associée. La clause **FROM WINDOWS**, permet de dire que le login existe dans le système d'exploitation Windows. Les deux options suivantes permettent quant à elles de choisir, et la base par défaut de la connexion, et la langue par défaut de cette même connexion.

Il est important de noter que les instructions **UPDATE** et **DELETE** peuvent être appliquées dans le cas d'un changement de propriété de connexion ou d'une suppression de connexion.

Exemple :

Création d'un compte intégré SQL Server pour l'utilisateur Windows **Util2** de la machine locale ou du domaine **03112-375** (les options de stratégie ne peuvent être définies, car elles sont définies au niveau Windows) sur la base **master**.

```
CREATE LOGIN [03112-375\Util2]
FROM WINDOWS
WITH DEFAULT_DATABASE = master
```

Idem, mais sur la base Papyrus :

```
CREATE LOGIN [03112-375\Util2]
FROM WINDOWS
WITH DEFAULT_DATABASE = Papyrus
```

En mode de sécurité mixte :

L'instruction **CREATE LOGIN** annonce à SQL Server qu'une nouvelle connexion va être créée. On donne impérativement un nom à cette connexion.

```
CREATE LOGIN nom_connexion
FROM PASSWORD='monmotdepasse'
Motdepasse HASHED MUST_CHANGED,
DEFAULT_DATABASE=Master,
DEFAULT_LANGUAGE=langue,
CHECK_EXPIRATION=ON,
CHECK_POLICY=ON,
CREDENTIAL=Nom
```

La clause **FROM** permet de donner plusieurs propriétés à cette connexion, qui sont les suivantes :

- **PASSWORD** : Permet de préciser un mot de passe. L'option **HASHED**, permet de hacher le mot de passe lors de son stockage en fonction de la chaîne de caractères précisée avant de mot clé.

- **DEFAULT_DATABASE** : Permet de préciser le nom de la base de données par default.

- **DEFAULT_LANGUAGE** : Permet de préciser un langage par défaut.

- **CHECK_EXPIRATION** : À **OFF** par défaut. Il n'est possible d'activer cette option que si **CHECK_POLICY** est aussi activé. Elle permet d'appliquer la politique de changement des mots de passe défini sur le serveur.

- **CHECK_POLICY** : A **ON** par défaut, cette option permet de récupérer au niveau serveur, les options définies pour la politique de sécurité.

- **CREDENTIAL** : Permet de relier la connexion à un credential créé auparavant. Nous verrons par la suite ce qu'est un credential.

On peut également utiliser la procédure stockée **sp_addlogin** qui permet à un utilisateur de se connecter à une instance de SQL Server à l'aide de l'authentification SQL Server.

```
EXEC sp_addlogin 'Utilisateur', 'motdepasse', 'basededonnees'
```

Exemple : Nous créons l'« Utilisateur1 » avec le mot de passe « pwd » sur la base de données « papyrus ».

```
EXEC sp_addlogin 'Utilisateur1', 'pwd', 'Papyrus'
```

Modification des connexions à SQL Server

Vous pouvez modifier les connexions par l'interface avec les écrans vus précédemment ou en utilisant du code T-SQL :

Activation d'une connexion désactivée. L'exemple suivant active la connexion Mary5.

```
ALTER LOGIN Mary5 ENABLE;
```

Modification du mot de passe d'une connexion. L'exemple suivant remplace le mot de passe de la connexion Mary5 par un mot de passe fort.

```
ALTER LOGIN Mary5 WITH PASSWORD = '<enterStrongPasswordHere>';
```

Modification du nom d'une connexion. L'exemple suivant remplace le nom de connexion Mary5 par John2.

```
ALTER LOGIN Mary5 WITH NAME = John2;
```

Mappage d'une connexion sur une information d'identification. L'exemple suivant mappe la connexion John2 aux informations d'identification Custodian04.

```
ALTER LOGIN John2 WITH CREDENTIAL = Custodian04;
```

Mappage d'une connexion à des informations d'identification EKM (Extensible Key Management). L'exemple suivant mappe la connexion Mary5 aux informations d'identification EKM EKMProvider1.

```
ALTER LOGIN Mary5
ADD CREDENTIAL EKMProvider1;
GO
```

Déverrouillage d'une connexion. Pour déverrouiller une connexion SQL Server, exécutez l'instruction suivante, en remplaçant * * * * par le mot de passe de compte souhaité.

```
ALTER LOGIN [Mary5] WITH PASSWORD = '****' UNLOCK ;
GO
```

Pour déverrouiller une connexion sans modifier le mot de passe, désactivez la stratégie de contrôle, puis réactivez-la.

```
ALTER LOGIN [Mary5] WITH CHECK_POLICY = OFF;
ALTER LOGIN [Mary5] WITH CHECK_POLICY = ON;
GO
```

Modification du mot de passe d'une connexion à l'aide de **HASHED**. L'exemple suivant modifie le mot de passe de la connexion TestUser en une valeur déjà hachée.

```
ALTER LOGIN TestUser WITH
PASSWORD = 0x01000CF35567C60BFB41EBDE4CF700A985A13D773D6B45B90900 HASHED;
GO
```

Suppression des connexions à SQL Server

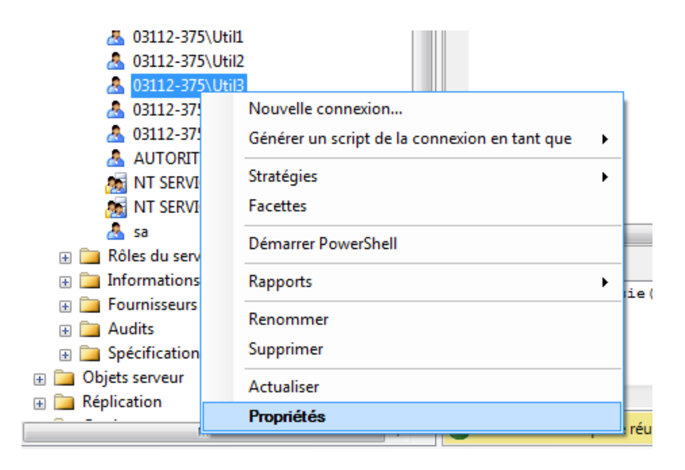
Vous pouvez modifier les connexions par l'interface avec les écrans vus précédemment ou en utilisant du code T-SQL.

Gestion des utilisateurs de base de données

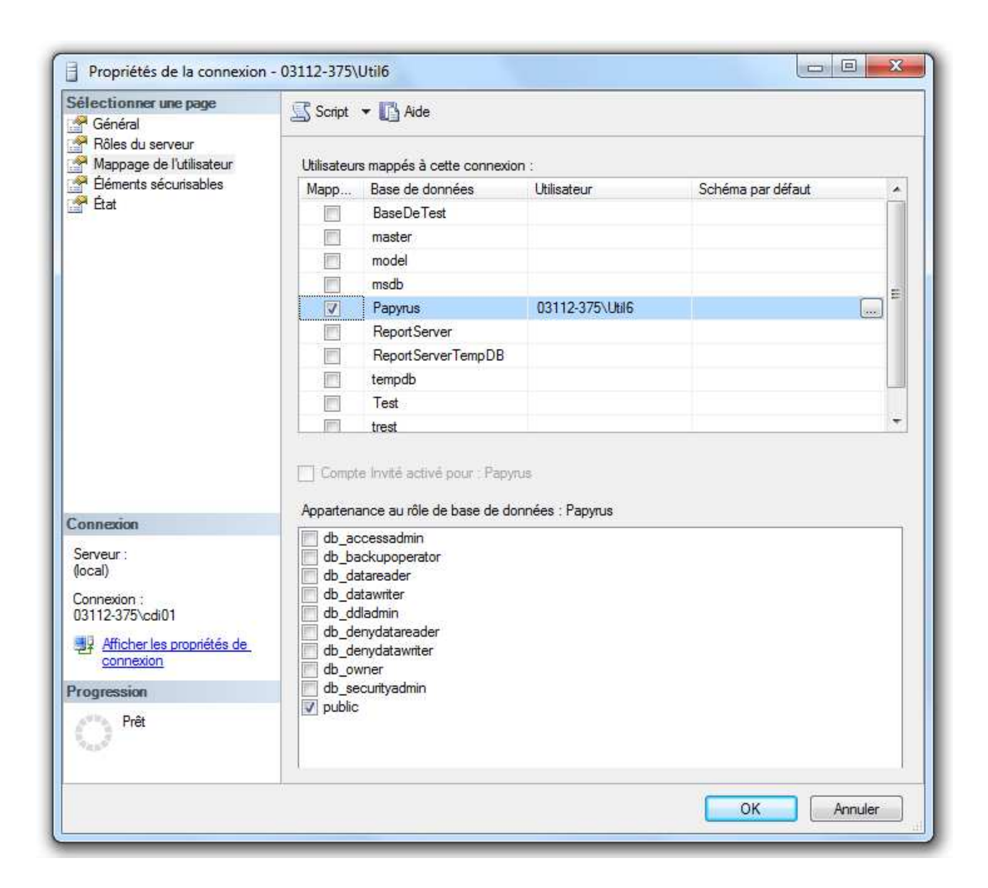
Après avoir créé un compte (identité de serveur) pour la connexion au serveur SQL, il convient de lui accorder des droits d'accès à une ou plusieurs bases de données. Une connexion ne pourra exécuter des opérations sur une base que s'il existe un compte utilisateur défini sur cette base, associé à la connexion.

En utilisant l'interface

Développez alors le nœud Sécurité et placez-vous sur le nœud Utilisateurs. Cliquez droit sur l'utilisateur puis « **Propriétés** ».



Dans l'onglet « **Mappage de l'utilisateur** », définissez les bases de données où l'utilisateur aura accès. Il est, entre autres, possible d'attacher des schémas à cet utilisateur ainsi que des rôles de base de données.



Par le code

En Transact SQL, l'instruction permettant de créer des utilisateurs et de les mapper aux différentes connexions est **CREATE USER**. Voici la syntaxe détaillée de cette instruction :

```
CREATE USER nom
FOR LOGIN connexion
CERTIFICATE nomcertificat
ASYMETRIC KEY nomcle
WITH DEFAULT_SCHEMA = nomschema
```

- **FOR LOGIN** : Définit la connexion où sera mappé l'utilisateur.

- **CERTIFICATE** : Définis le certificat à utiliser. Ne peut pas s'utiliser si une connexion est utilisée dans la clause **FOR LOGIN**.

- **ASYMETRIC KEY** : Définis la clé asymétrique à utiliser.

- **WITH DEFAULT SCHEMA** : Nom du schéma de base de données à donner à l'utilisateur.

Il existe des procédures stockées pour créer des utilisateurs et leur donner des droits sur les bases de données qui sont **sp_grantdbaccess** et **sp_adduser**. Celles-ci sont maintenues pour des raisons de compatibilité ascendante, cependant, il est conseillé de ne plus les utiliser, car elles sont vouées à disparaître dans les prochaines versions de SQL Server.

Exemple :

Création de l'utilisateur Camille dans la base de données AdventureWorks avec le schéma Sales.

```
USE AdventureWorks
CREATE USER Camille FOR LOGIN Camille WITH DEFAULT_SCHEMA = Sales
```

Création de l'utilisateur Util5 dans la base de données Papyrus avec le schéma vente.

```
USE Papyrus
CREATE USER Util5 FOR LOGIN [03112-375\Util5] WITH DEFAULT_SCHEMA = vente
```

On peut également utiliser la procédure stockée **sp_adduser** qui ajoute un nouvel utilisateur dans la base de données active. Par exemple, on ajoute l'utilisateur nommé « Gestionnaire » dont la connexion est « Utilisateur ».

```
sp_adduser 'Utilisateur', 'Gestionnaire'
```

sp_adduser créera également un schéma qui aura le nom de l'utilisateur. Une fois qu'un utilisateur a été ajouté, utilisez les instructions **GRANT**, **DENY** et **REVOKE** afin de définir les autorisations contrôlant les activités effectuées par l'utilisateur.

Modification des utilisateurs de base de données

En utilisant l'interface

Pour modifier un utilisateur de base de données via l'interface graphique, il vous suffit de déployer l'ensemble des nœuds qui mène à cet utilisateur dans l'explorateur d'objets, afficher le menu contextuel et sélectionner **propriétés**. Vous arriverez alors directement sur la fenêtre de création d'un utilisateur, ou certains champs seront remplis. Il vous suffira de changer les informations que vous désirez modifier.

Par le code

Pour modifier un utilisateur avec du code, nous allons utiliser l'instruction **ALTER USER**. Voici la syntaxe de modification des propriétés d'un utilisateur.

```
ALTER USER nomutilisateur
WITH NAME=nouveaunom,
DEFAULT_SCHEMA=nouveauschema
```

Exemples :

Modification du nom d'un utilisateur de base de données. L'exemple suivant modifie le nom de l'utilisateur de base de données Mary5 en Mary51.

```
USE AdventureWorks2008R2;
ALTER USER Mary5 WITH NAME = Mary51;
GO
```

Modification du schéma par défaut d'un utilisateur. L'exemple suivant modifie le schéma par défaut de l'utilisateur Mary51 en Purchasing.

```
USE AdventureWorks2008R2;
ALTER USER Mary51 WITH DEFAULT_SCHEMA = Purchasing;
```

```
GO
```

Modification du schéma de l'utilisateur Util5.

```
USE Papyrus
ALTER USER [03112-375\Util5] WITH DEFAULT_SCHEMA = vente;
GO
```

Suppression des utilisateurs de base de données

En utilisant l'interface

La procédure est la même que pour la modification. Simplement, au lieu d'afficher le menu contextuel et de sélectionner **propriété**, nous allons sélectionner **supprimer**.

Par le code

Pour supprimer un utilisateur, nous allons utiliser l'instruction **DROP USER**. Voici la syntaxe de cette instruction :

```
DROP USER nomutilisateur
```

Gestion des droits

Les droits sont les autorisations qui vont nous permettre de travailler avec notre base de données. Ils sont organisés de façon hiérarchique par rapport aux éléments sécurisables du serveur.

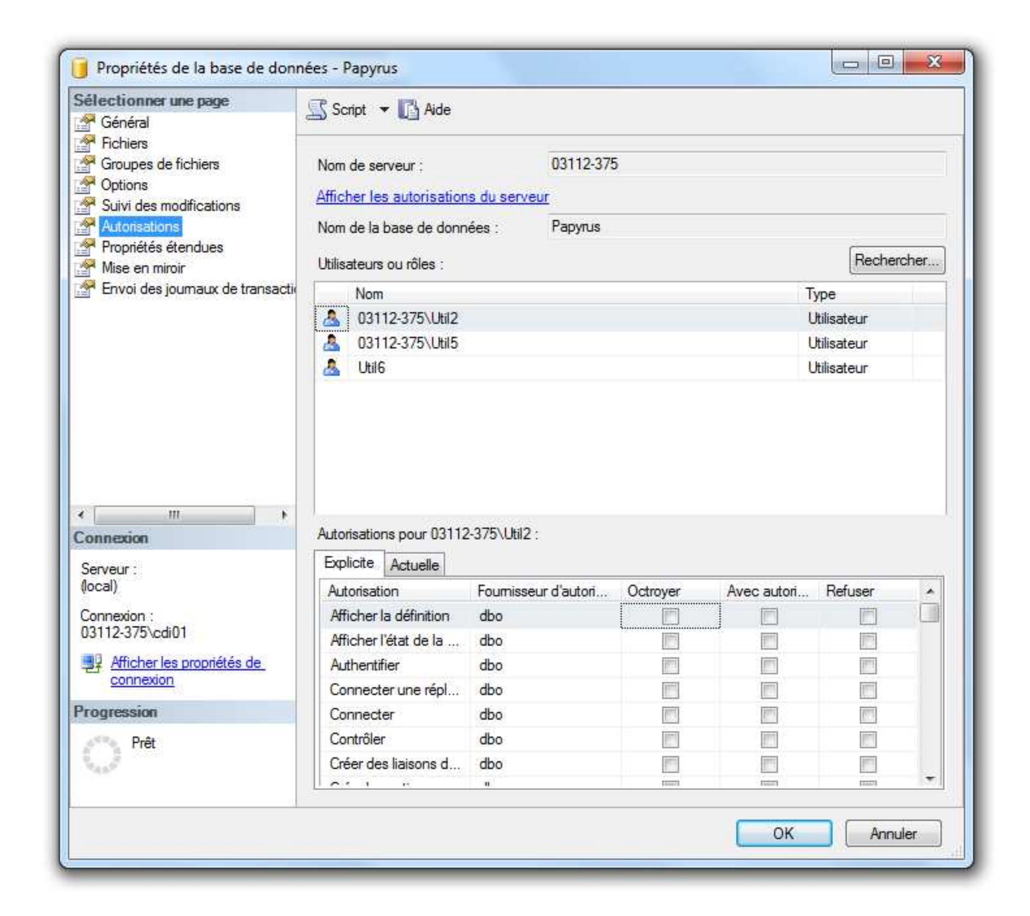
L'attribution des droits peut être faite à tous les niveaux, que ce soit au niveau serveur, au niveau base de données ou encore directement sur les schémas ou les objets. Par conséquent, ils peuvent être accordés soit à un utilisateur, soit à une connexion. Il est possible de gérer ces permissions grâce à 3 instructions simples dans SQL Server : **GRANT**, **DENY**, **REVOKE**. **GRANT** permet de donner un privilège, **REVOKE** permet de le retirer si celui-ci a été donné auparavant et **DENY** permet de l'interdire même si il a été donné au travers d'un rôle.

Droits d'instruction

Ces droits correspondent aux droits qui permettent de créer (mettre à jour, supprimer) de nouveaux objets dans la base. Les utilisateurs qui possèdent de tels droits sont donc capables de créer leurs propres tables… Voici les principaux droits disponibles : **CREATE DATABASE**, **CREATE TABLE**, **CREATE FUNCTION**, **CREATE PROCEDURE**, **CREATE VIEW**, **BACKUP DATABASE**, **BACKUP LOG**… Nous allons maintenant apprendre à donner, retirer ou interdire des droits.

En utilisant l'interface

Avec SSMS, on peut administrer ces droits via la fenêtre propriété de la base de données concernée. Pour cela, rendez-vous sur le menu Autorisations de cette même page.



Pour ajouter des droits à un utilisateur, sélectionnez l'utilisateur dans le premier champ, où nous avons notre utilisateur **Guest**, et octroyez-lui, enlevez-lui ou refusez-lui des permissions en cochant ou décochant les checkbox en conséquence.

Par le code

Pour ajouter, supprimer ou interdire un droit à un utilisateur de base de données, nous allons utiliser les instructions **GRANT**, **DENY** et **REVOKE** comme énoncé précédemment. Voici la syntaxe de don, suppression ou interdiction de droits :

```
--Ajout :
GRANT CREATE TABLE TO Guest
--Suppression :
REVOKE CREATE TABLE FROM Guest
--Interdiction :
DENY CREATE TABLE TO Guest
```

Droits d'utilisation

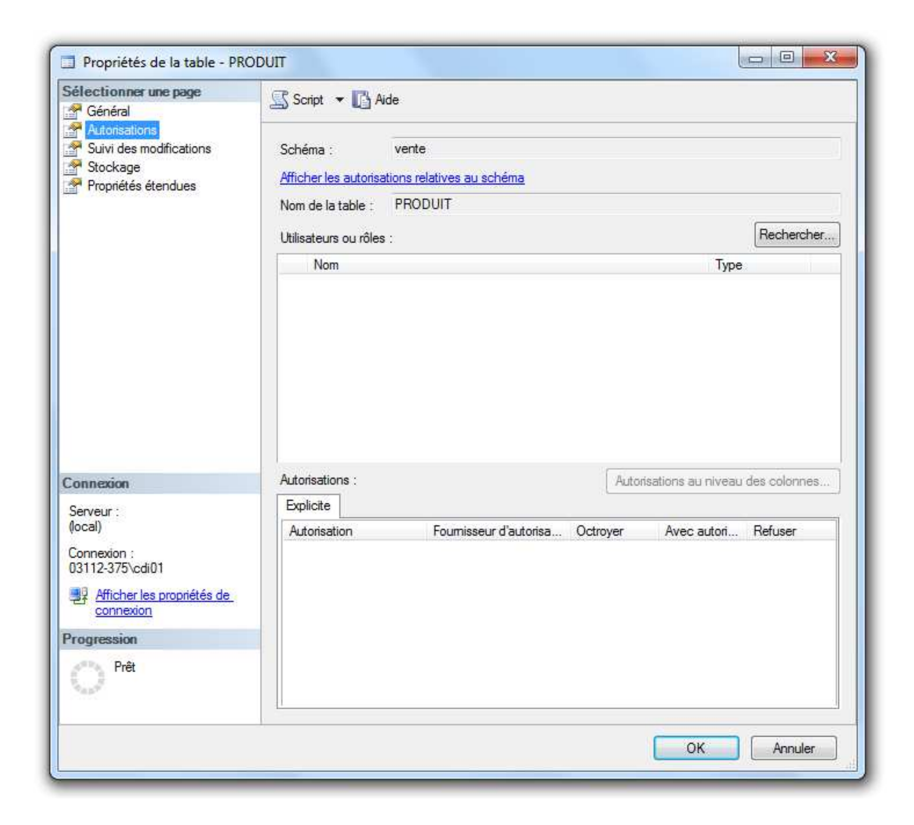
Les droits d'utilisation permettent de déterminer si un utilisateur possède les droits pour travailler sur les objets, par exemple lire les informations, insérer des données… Dans le cas

général, c'est le propriétaire de l'objet qui définit les droits d'utilisation. En ce qui concerne les droits de lecture et de mise à jour des données, le propriétaire possède la faculté de choisir quelles colonnes l'utilisateur peut lire ou mettre à jour. Voici la liste des principaux droits d'utilisation : **INSERT**, **UPDATE**, **SELECT**, **DELETE**, **EXECUTE**.

En utilisant l'interface

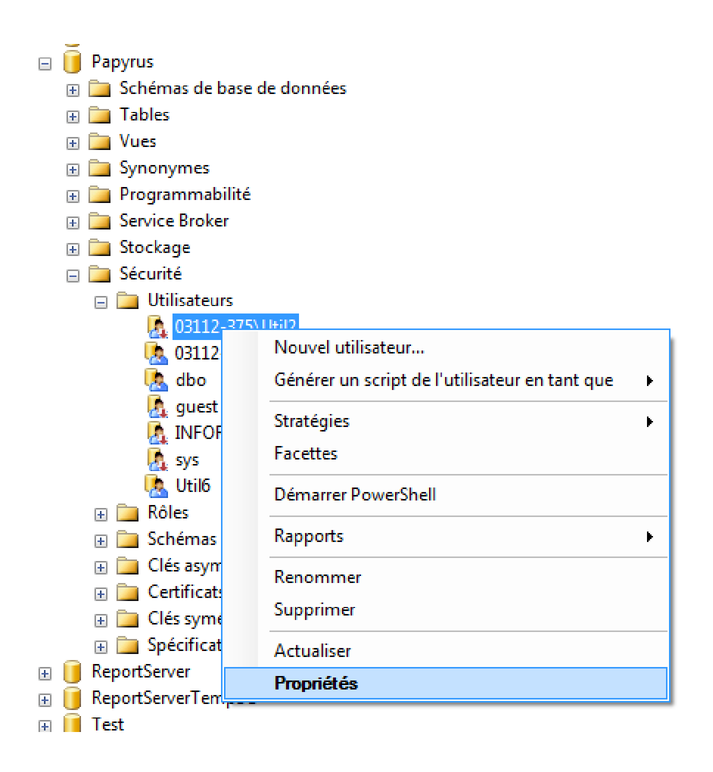
Les droits d'utilisation peuvent être gérés à deux niveaux dans SQL Server, au niveau utilisateur et au niveau objet. Dans SSMS, pour les deux niveaux cités, la gestion de ces droits se fait grâce à la fenêtre de propriété. Voici les deux cas de fenêtre dans lesquelles vous pouvez changer les droits d'utilisation.

- <u>Au niveau objet</u> :

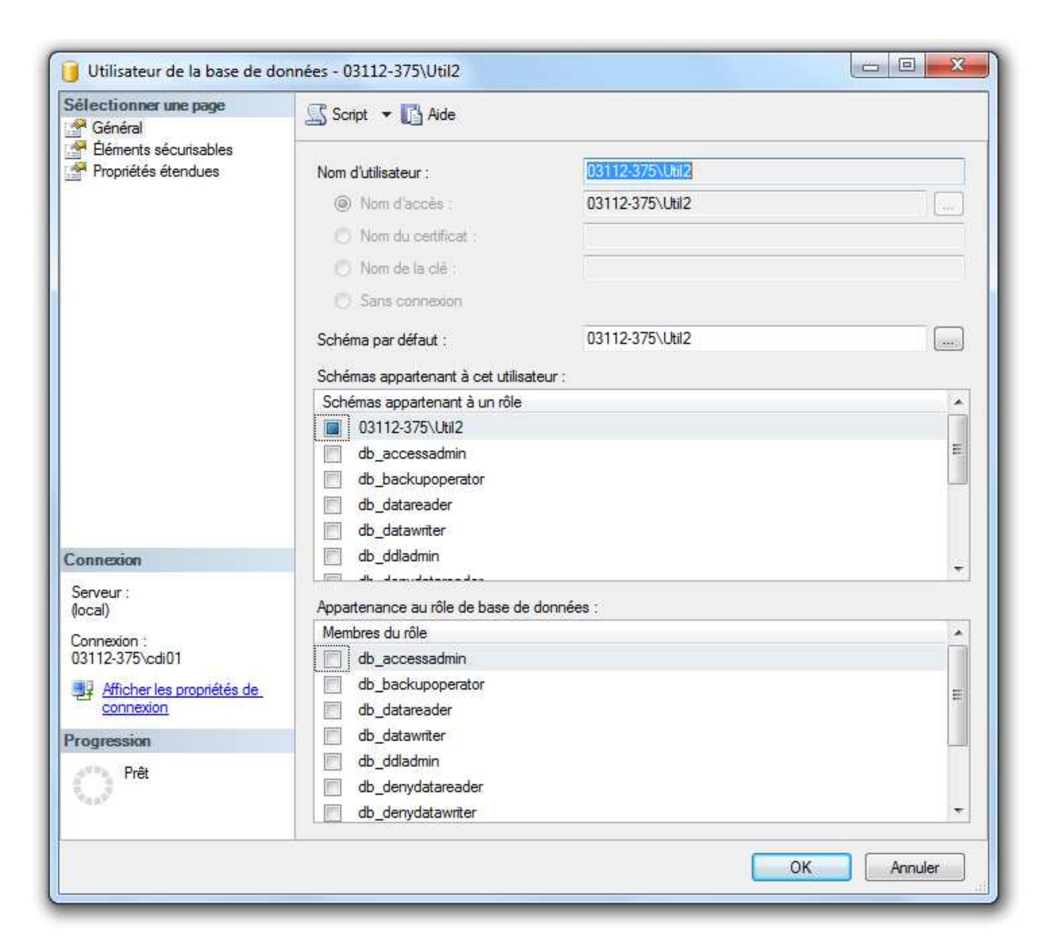


- <u>Au niveau utilisateur</u> :

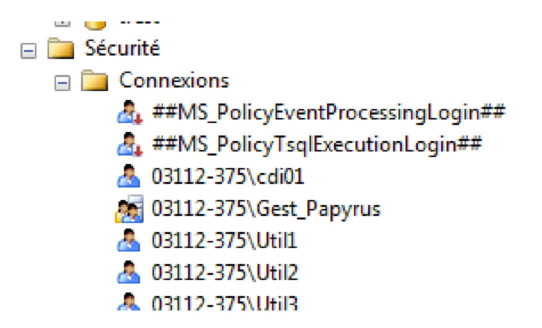
Cliquez droit puis « **Propriétés** » sur l'utilisateur du dossier des utilisateurs de votre base de données.



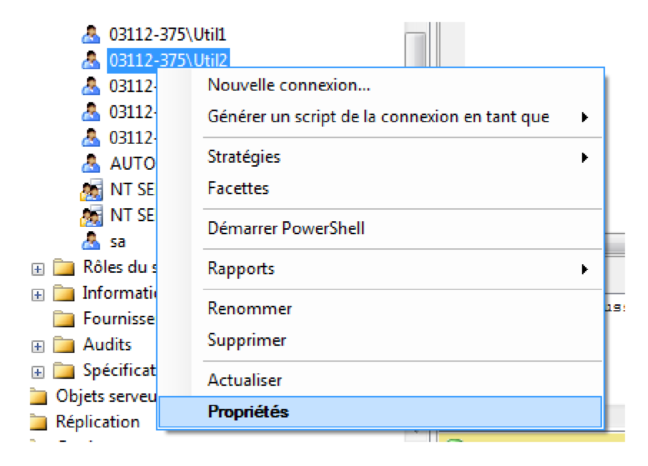
La fenêtre suivante apparaît alors.



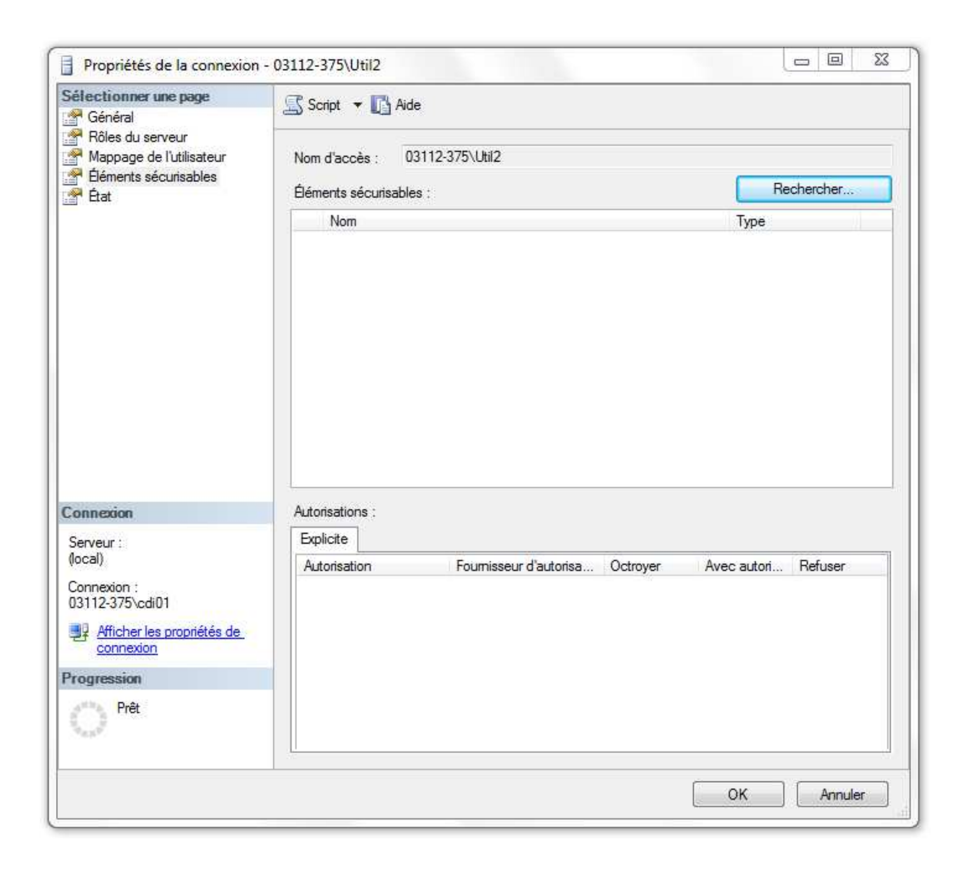
Si vous voulez intervenir au niveau de toutes les tables dans la base, vous devez vous rendre sur le dossier des « **Connections** » au niveau « **Sécurité** » de toutes les bases.



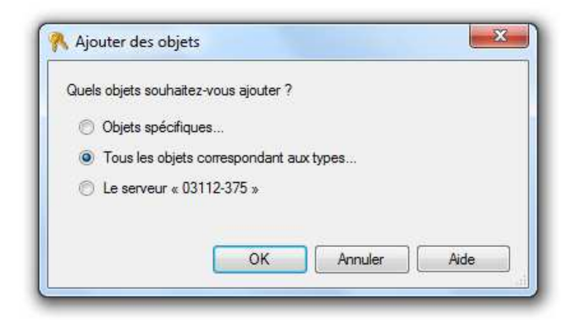
Cliquez droit sur l'utilisateur concerné puis « **Propriété** ».



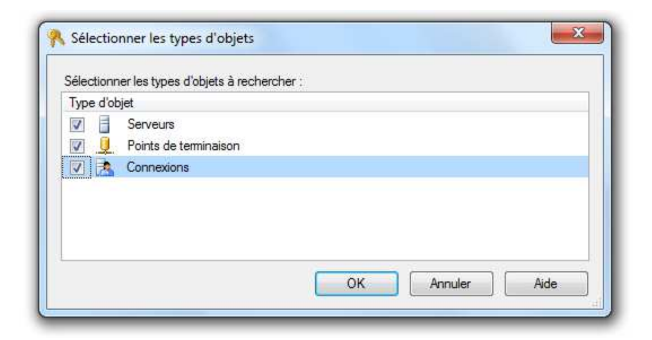
Sélectionnez l'onglet « **Elements sécurisables** » en allant, par exemple rechercher tous les éléments de type table, et en affectant les autorisations souhaitées.



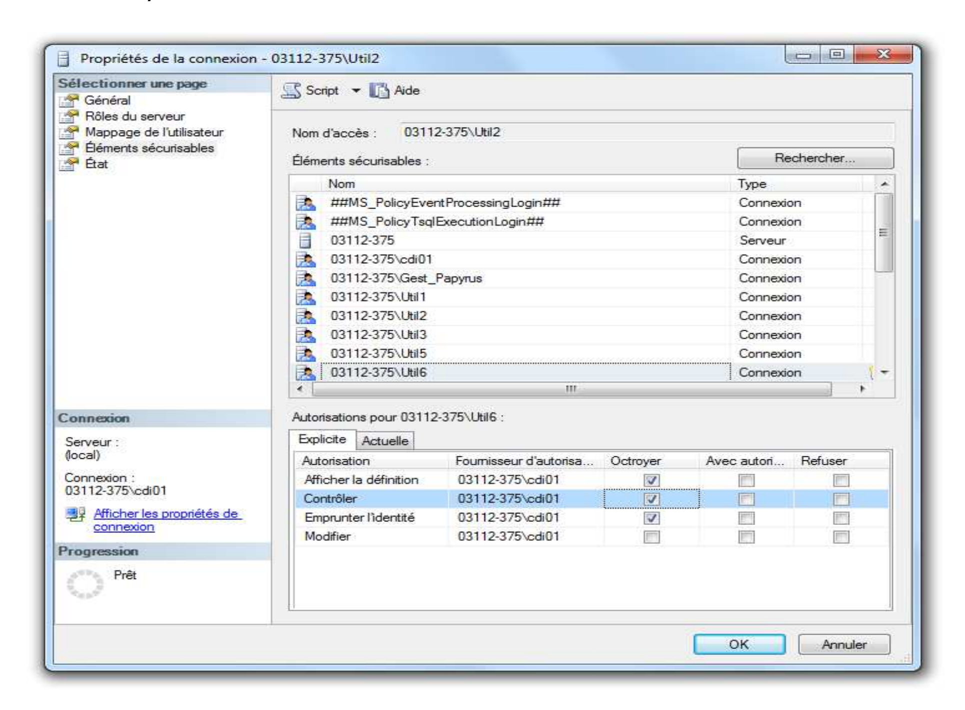
Cliquez sur « **Rechercher…** » puis ajouter les objets souhaités.



Vous aurez notamment à sélectionner les types d'objets correspondant à votre recherche. Puis cliquez sur « **Ok** ».



Il vous reste à octroyer les droits en cochant les cases.



Par le code

Pour donner des droits d'utilisation ou les retirer avec du code T-SQL, nous allons retrouver les trois mots clés **GRANT**, **DENY** et **REVOKE**. La syntaxe en revanche sera changeante. La voici :

```
-- Possibilité de lire dans toutes les tables de la base
GRANT SELECT TO [03112-375\Util2]

-- Possibilité d'insérer dans toutes les tables de la base
GRANT UPDATE TO [03112-375\Util2]

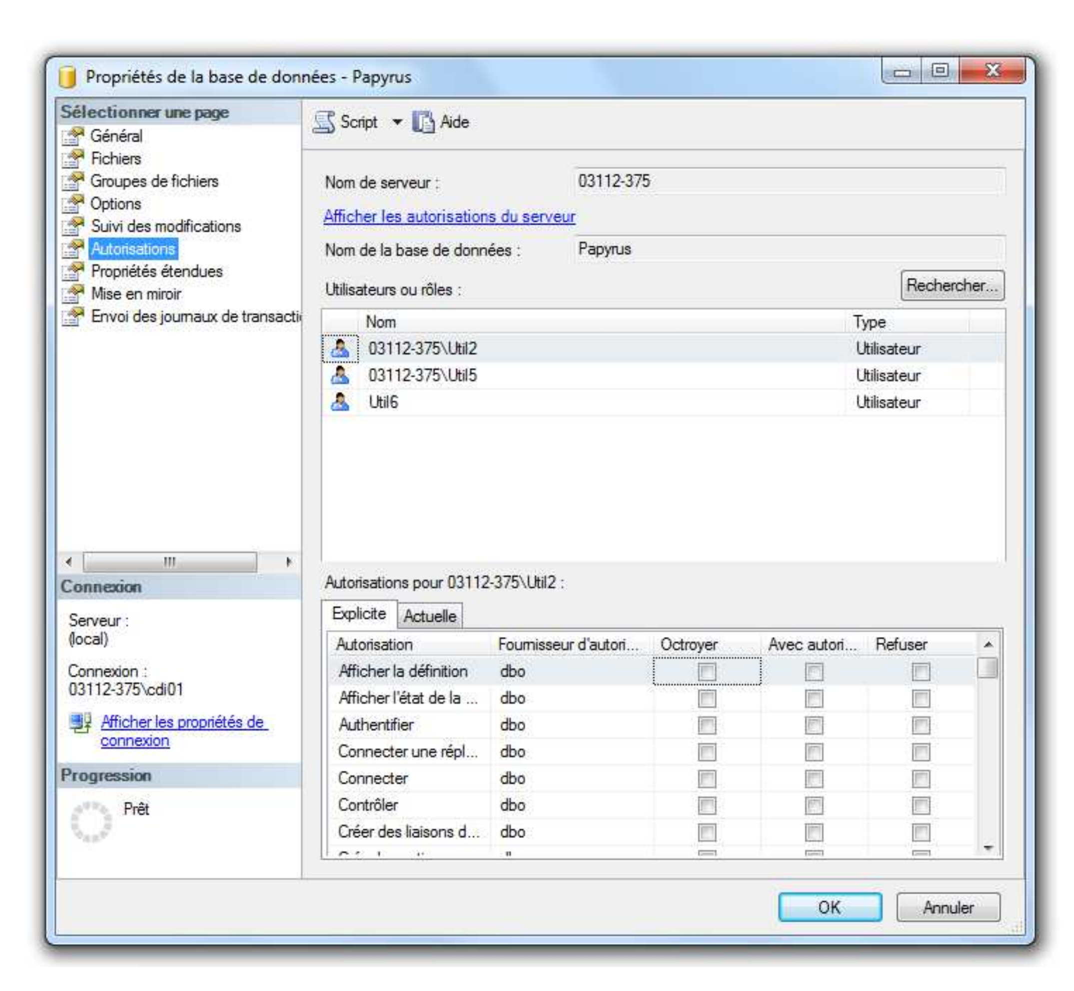
-- Interdire la suppression sur toutes les tables.
DENY DELETE TO [03112-375\Util2]
```

Droits au niveau base de données

Les droits au niveau des bases de données vont donner des droits aux utilisateurs qui ne seront valables que sur une base de données précise. Au niveau base de données, il est possible de donner des droits à un utilisateur, à un schéma, à une assembly ou encore à un objet service broker. Ces droits peuvent être attribués de plusieurs manières, soit par du code Transact SQL, soit par les propriétés de la base de données.

En utilisant l'interface

Pour accorder des droits, vous procéderez de la façon suivante. Déroulez la totalité des nœuds qui mènent à votre base de données, affichez le menu contextuel de cette base de données en effectuant un clic droit et sélectionnez **propriété**. Vous aboutissez sur la fenêtre suivante (pensez à vous rendre dans la partie **Autorisations**) :



Il vous suffit d'ajouter des utilisateurs dans la première partie nommée **Utilisateurs ou rôles** et de leur donner des droits dans la seconde partie nommée **Autorisations**. Cliquez enfin sur « **OK** » pour valider vos choix.

Par le code

```
GRANT permissiondebasededonnées
TO utilisateur
AS groupeourole
```

- Permissiondebasededonnées : La ou les permissions de base de données à accorder à l'utilisateur de base de données.

- **Utilisateur** : L'utilisateur qui va recevoir les permissions de base de données précisées après l'instruction **GRANT**.

- **Groupeourole :** Constitue le contexte de sécurité qui va nous permettre d'accorder les privilèges.

Les rôles

On peut dire que les rôles sont des sortes de groupements de droits. Il existe trois niveaux d'actions pour les rôles : Server, Base de données et Application. L'utilité des rôles réside dans le fait qu'il est plus simple d'attribuer des droits à des rôles puis d'attribuer des rôles à des utilisateurs ou des connexions plutôt que d'attribuer directement des droits à ces derniers. Il faut donc retenir que les rôles sont des ensembles de droits qui portent un nom et qu'on peut les attribuer aux utilisateurs.

Pour faciliter la gestion des droits, SQL Server propose des droits dits fixes. En effet, ils sont prédéfinis et donc non modifiables. Ils sont définis à deux niveaux : Server et Base de données. En revanche, les rôles définis par l'utilisateur de SQL Server atteignent deux niveaux différents : Base de données et Application. Au final, un utilisateur de base de données peut avoir accès à des droits de quatre manières différentes : Grâce aux droits de la connexion qu'il utilise, grâce aux rôles fixes, grâce aux rôles créés par l'utilisateur et enfin grâce aux droits qui lui ont été directement affectés.

Il existe un rôle qui est public et qui est attribué à tous les utilisateurs de base de données et à toutes les connexions. Ce rôle ne peut pas être enlevé à aucun des utilisateurs. En revanche, il peut être modifié. Il faut simplement prendre en compte que TOUS les utilisateurs auront les droits ajoutés à public.

Les rôles prédéfinis

<u>Rôles serveur</u> :

- Sysadmin : Administrateur du serveur.
- Serveradmin : Permet de configurer les paramètres niveau serveur.
- Setupadmin : Permet d'exécuter certaines procédures stockées et d'ajouter des serveurs liés.
- Securityadmin : Permet de gérer les connexions serveur.
- Processadmin : Permet de gérer les traitements au sein de SQL Server.

- **Dbcreator :** Permet de créer ou modifier des bases de données.
- **Diskadmin :** Permet de gérer les fichiers sur le disque.
- **Bulkadmin :** Permet d'exécuter l'instruction **BULK INSERT**.

Rôles base de données :

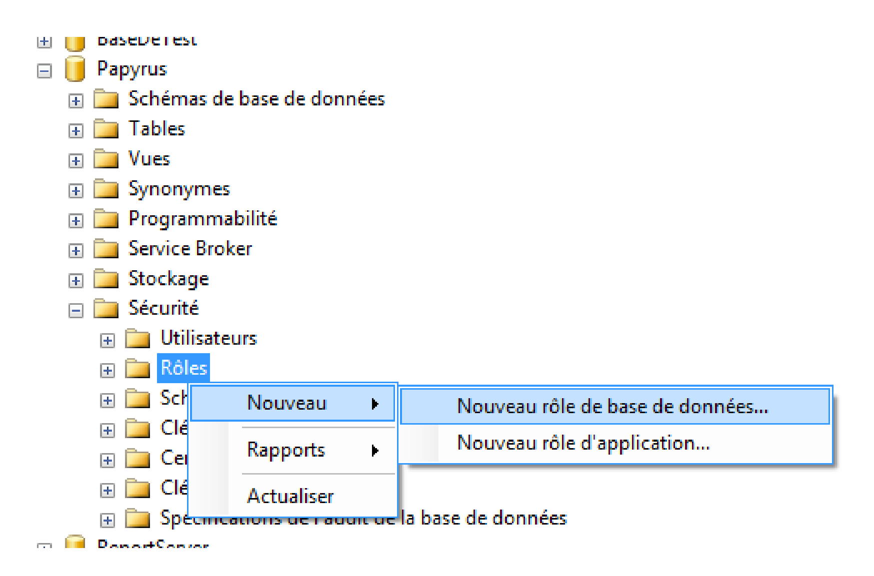
- **Db_owner :** Équivalent à propriétaire base de données.
- **Db_accessadmin :** Permet d'ajouter et supprimer des utilisateurs de base de données.
- **Db_datareader :** Permet d'utiliser l'instruction **SELECT**.
- **Db_datawriter :** Permet les instructions **INSERT**, **UPDATE** et **DELETE**.
- **Db_ddladmin :** Permet les opérations sur les objets de base de données.
- **Db_securityadmin :** Permet de gérer les éléments de sécurité sur la base de données.
- **Db_backupoperator :** Permet l'utilisation des backups.
- **Db_denydatareader :** Interdit l'instruction **SELECT**.
- **Db_denydatawriter :** Interdit l'écriture sur la base de données.

Les rôles définis par l'utilisateur

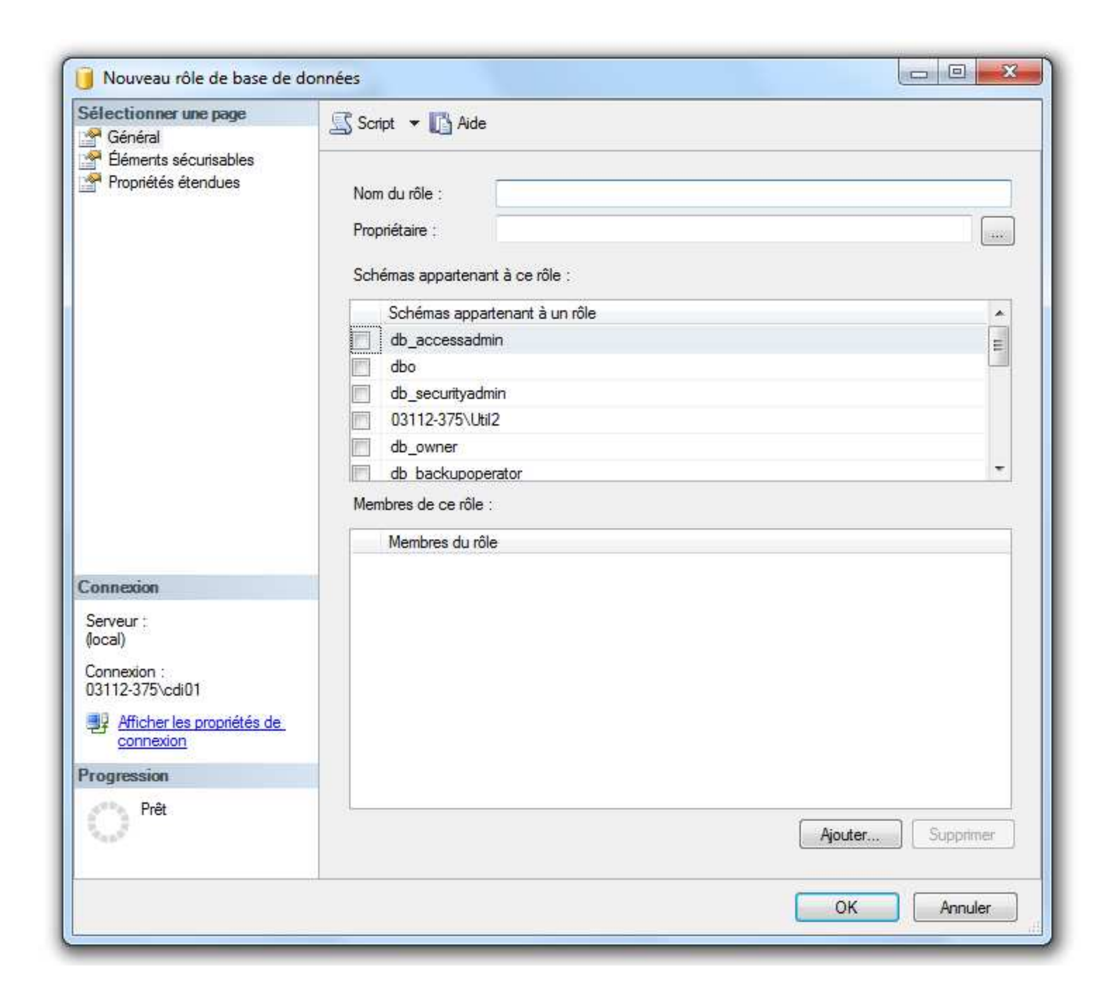
Il est possible de définir ses propres rôles afin de faciliter l'administration des droits dans SQL Server. Logiquement, on créera un rôle dit personnaliser lorsque plusieurs utilisateurs doivent avoir les mêmes droits et que ces droits n'existent pas dans les rôles prédéfinis. Les rôles peuvent être accordés soit directement à un utilisateur, soit à un autre rôle.

En utilisant l'interface

Pour créer un nouveau rôle, procédez de la manière suivante. Déployez successivement le nœud de votre base de données puis le nœud sécurité. Affichez alors le menu contextuel du nœud rôles et sélectionnez « **Nouveau rôle** ».



Vous obtenez cette nouvelle fenêtre :



Il vous suffit alors de préciser le nom et le propriétaire du rôle. En revanche, vous n'êtes pas obligé de préciser les autres informations de suite. Vous pouvez les modifier ultérieurement en vous rendant dans les propriétés du rôle voulu.

Par le code

Voici la syntaxe de création d'un rôle personnalisé :

```
CREATE ROLE nomdurole
AUTHORIZATION propriétaire
```

Nous allons bien entendu créer un rôle grâce à l'instruction **CREATE ROLE**. Il est alors nécessaire de préciser le nom du schéma après cette instruction pour assurer son unicité au sein de la base de données sur le serveur. On peut enfin préciser un propriétaire grâce à la clause **AUTHORIZATION**.

Exemple : Dans un premier temps, on crée un rôle nommé **agentsresa** qui n'autorise que la lecture (SELECT), la mise à jour (UPDATE), l'insertion (INSERT) et la suppression (DELETE) sur les tables (RESERVATION, OCCUPANT, REGLEMENT, ADHERENT), et on ajoute un utilisateur à ce rôle.

```
CREATE ROLE agentsresa;
GRANT SELECT, INSERT, UPDATE, DELETE
ON Gestion.RESERVATION
```

```
TO agentsresa;
GRANT SELECT, INSERT, UPDATE, DELETE
ON Gestion.OCCUPANT
TO agentsresa;
GRANT SELECT, INSERT, UPDATE, DELETE
ON Gestion.REGLEMENT
TO agentsresa;
GRANT SELECT, INSERT, UPDATE, DELETE
ON Gestion.ADHERENT
TO agentsresa;
```

L'ajout d'un utilisateur affecté à des rôles prédéterminés se fait en deux temps :

. Création de la connexion
. Ajout d'un utilisateur et de son rôle

Nous pouvons utiliser les procédures stockées du système. Création d'un utilisateur nommé Gestionnaire1 dont la connexion est Utilisateur1, le mot de passé pwd et possédant le rôle agentsresa :

```
USE Tourismes
GO
EXEC sp_addlogin 'Utilisateur1', 'pwd', 'Tourismes'
GO
sp_adduser 'Utilisateur1', 'Gestionnaire1', 'agentsresa'
GO
```

Le Langage DML

Les variables locales

On définit une variable locale à l'aide d'une instruction **DECLARE**. Les instructions **SET** ou **SELECT** peuvent être utilisées pour assigner une valeur. On l'utilise dans le cadre de l'instruction, du lot ou de la procédure dans laquelle elle a été déclarée. Le nom d'une variable locale commence toujours par **@**.

Exemple 1 : Liste de tous les employés dont le nom commence par une chaîne de caractères, donnée (B).

```
DECLARE @vrech char(2)
SET @vrech ='B%'

SELECT noemp, prenom, nom, dept FROM Employés
WHERE nom like @vRech
```

Exemple 2 : Renvoi du numéro d'employé le plus élevé.

```
DECLARE @vempId int
SELECT @vempId = max(noemp) FROM Employés
```

Si l'instruction **SELECT** renvoie plusieurs lignes, la valeur affectée à la variable est celle correspondant à la dernière ligne renvoyée.

```
DECLARE @vempId int
SELECT @vempId = noemp FROM Employés
```

Les variables système

TRANSACT-SQL propose de nombreuses variables qui renvoient des informations. Certaines d'entre elles demandant des paramètres en entrée. Elles se distinguent des variables utilisateur par le double **@** : la syntaxe d'utilisation de la plupart des fonctions est **@@nom_fonction**.

La variable **@@TRANCOUNT** renvoie le nombre de transactions ouvertes.

La variable **@@ROWCOUNT** renvoie une valeur représentant le nombre de lignes affectées par la dernière requête effectuée.

La variable **@@ERROR** contient le numéro de l'erreur de la dernière instruction TRANSACT-SQL exécutée. Elle est effacée et réinitialisée chaque fois qu'une instruction est exécutée. Si l'instruction s'est exécutée avec succès, **@@ERROR** renvoie la valeur 0. On peut utiliser la fonction **@@ERROR** pour détecter un numéro d'erreur particulier ou sortir d'une procédure stockée de manière conditionnelle.

La variable **@@IDENTITY** renvoie la valeur de la dernière valeur d'identité insérée

La forme conditionnelle IF

C'est la structure alternative qui permet de tester une condition et d'exécuter une instruction si le test est vrai.

Exemple 1 : Si la moyenne des salaires est inférieure à 1500, l'augmentation est nécessaire.

```
IF (Select AVG(salaire) from EMPLOYES ) < 1500
BEGIN
UPDATE EMPLOYES SET Salaire = salaire * 12
SELECT noemp, salaire from EMPLOYES
PRINT 'Augmentation effectuée'
END
```

Exemple 2 : Vérification que le département 'E21' comporte au moins un salarié avant de le supprimer

```
IF EXISTS (SELECT Nodept FROM Depart WHERE Nodept ='E21')
PRINT '*** impossible de supprimer le client ***'
ELSE
BEGIN
DELETE Depart WHERE Nodept = 'E21'
PRINT '*** Client supprimé **'
END
```

On notera la présence obligatoire de la structure **BEGIN … END** dès lors que le nombre d'instructions a exécuté est supérieur à 1.

La fonction CASE

La fonction **CASE** évalue une liste de conditions et renvoie la valeur de l'expression de résultat correspondant à la condition sélectionnée.

Exemple 1 : Affichage du salaire sous forme d'un commentaire texte basé sur une fourchette de salaire.

```
SELECT nom, 'Catégorie de Salaire' =
CASE
WHEN salaire IS NULL THEN 'Non divulgué !'
WHEN salaire < 1500 THEN 'Agent de maîtrise'
WHEN salaire >= 1500 and salaire < 2000 THEN 'Cadre'
ELSE 'PDG!'
END
FROM Employés ORDER BY salaire
```

Exemple 2 : Affichage du salaire sous forme d'un commentaire texte basé sur une fourchette de salaire.

```
SELECT nom, 'Catégorie de Salaire ' =
CASE
WHEN salaire IS NULL THEN 'Non divulgué !'
WHEN salaire < 1500 THEN 'Agent de maîtrise'
WHEN salaire >= 1500 and salaire < 2000 THEN 'Cadre'
ELSE 'PDG!'
END
FROM Employés ORDER BY salaire
```

La boucle WHILE

C'est la structure répétitive qui permet d'exécuter une série d'instructions tant qu'une condition est vraie. L'exécution des instructions de la boucle peut être contrôlée par les instructions **BREAK** et **CONTINUE** (rarement utilisées).

L'instruction **BREAK** permet de sortir de la boucle. **CONTINUE** permet de repartir à la première instruction de la boucle.

Exemple 1 : Utilisation du **While**

```
DECLARE @Compteur int
SET @Compteur =1
WHILE @Compteur <= 10
BEGIN
INSERT.....
SET @compteur += 1
END
```

Exemple 2 : Utilisation du **While**

```
DECLARE @Compteur int = 1
WHILE @Compteur <= 10
BEGIN
INSERT .....
IF <cond>
BEGIN
... instructions ...
BREAK
END
SET @compteur += 1
END
```

L'instruction RETURN

Elle permet de terminer l'exécution en renvoyant éventuellement une variable entière, et retourner le contrôle à l'application appelante. Tout code après l'instruction **RETURN** ne sera pas exécuté.

L'instruction PRINT

Elle permet d'afficher un message.

La clause OUTPUT

SQL Server crée et gère automatiquement les tables **inserted** et **deleted** qui ont la même structure que la table sur laquelle porte la requête **INSERT**, **UPDATE**, **DELETE** ou **MERGE**.

Avant SQL Server 2005, on pouvait accéder à ces tables uniquement à partir d'un déclencheur. On peut désormais accéder à ces tables directement dans le cadre d'une instruction **INSERT**, **UPDATE**, **DELETE** ou **MERGE**, grâce à la clause **OUTPUT**.
La clause **OUTPUT** permet de travailler avec ces tables de 2 manières :

. Elle permet de retourner leur contenu directement à l'application, en tant qu'ensemble de résultats

. Elle permet d'insérer leur contenu dans une table ou variable de table.

Exemple 1 **:** Supprimer tous les employés de la table EMPLOYES et retourner à l'application l'ensemble des lignes supprimées.

```
DELETE FROM EMPLOYES OUTPUT deleted.*
```

Exemple 2 **:** Insérer une ligne dans la table EMPLOYES, et retourner à l'application le champ de la table **inserted** contenant le numéro d'employé.

```
DECLARE @Tajout table(num int )
INSERT INTO EMPLOYES (NOEMP, NOM, PRENOM, DEPT)
VALUES (00140,'REEVES','HUBERT','A00')
OUTPUT inserted.NOEMP INTO @Tajout
```

On pourra ensuite visualiser les données ajoutées en affichant le contenu de la table Tajout.

Exemple 3 : Supprimer les employés de la table EMPLOYES du département B00 et les renvoyer dans une variable table à l'application.

```
DECLARE @EmpTableB00 table (
NOEMP int NOT NULL,
NOM varchar(40) NOY NULL,
PRENOM varchar(20) NOT NULL,
SALAIRE int NULL)
DELETE EMPLOYES
OUTPUT deleted.NOEMP,
deleted.NOM,
deleted.PRENOM,
deleted.SALAIRE
INTO @EmpTableB00
WHERE DEPT ='B00'
-- Affichage de la table créée
SELECT NOEMP, NOM, PRENOM, SALAIRE FROM @EmpTableB00
```

Les messages d'erreurs

Pour chaque erreur, SQL Server produit un message d'erreur. La plupart de ces messages sont définis dans SQL Server, mais il est possible de définir ses propres messages grâce à la procédure stockée système **sp_addmessage**.

Tous les messages stockés à l'aide de **sp_addmessage** peuvent être affichés grâce à l'affichage catalogue **sys.messages**.

```
SELECT * FROM SYS.MESSAGES
```

Exemple 1 : Ajout d'un message simple mentionnant qu'un employé n'existe pas

```
EXECUTE sp_addmessage 50001, 16, 'Le code employé est inexistant',
'us_english'
EXECUTE sp_addmessage 50001, 16, 'Le code employé est inexistant'
```

Le message est d'abord ajouté en langue anglaise, puis en français.

Exemple 2 : Ajout d'un message simple mentionnant que l'employé {code} n'existe pas dans le service {nom}.

```
EXECUTE sp_addmessage 50002, 16, 'Le code employé %d est inexistant dans le
service %s', 'us_english'
EXECUTE sp_addmessage 50002, 16, 'Le code employé %1 ! est inexistant dans
le service %2 !'
```

La syntaxe exacte de l'expression de format de la variable prévoit l'espacement et la justification, le nombre maximum de caractères à prendre en compte.

Utilisation de NOCOUNT, EXISTS

NOCOUNT : Empêche le message indiquant le nombre de lignes affectées par une instruction Transact-SQL d'être renvoyé en tant que résultat.

```
SET NOCOUNT {ON | OFF}
```

Si **SET NOCOUNT** est activée (**ON**), le chiffre indiquant le nombre de lignes affectées par une instruction Transact-SQL n'est pas renvoyé. Si la valeur de **SET NOCOUNT** est définie sur **OFF**, ce chiffre est renvoyé.

EXISTS : Précise une sous-requête pour déterminer l'existence ou non de lignes.

Les fonctions utilisateur

Les fonctions utilisateurs sont de plusieurs types. Trois pour être précis. Il y a :

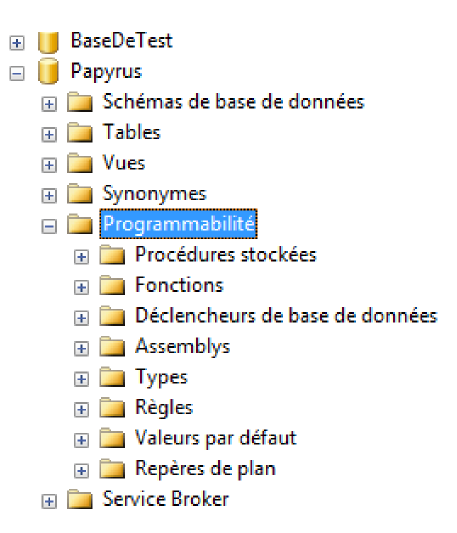
. Les fonctions scalaires
. Les fonctions tables en ligne
. Les fonctions tables multi-instructions.

Une fonction peut accepter des arguments, et ne peut retourner que deux types de données : une valeur scalaire ou une table.

- Les fonctions scalaires retournent, grâce au mot clé **RETURN**, une valeur scalaire. Tous les types de données peuvent être retournés par une fonction scalaire hors mis timestamp, table, cursor, text, ntext et image.

- Les fonctions table ne retournent comme résultat, qu'une table, qui est le résultat d'une instruction **SELECT**.

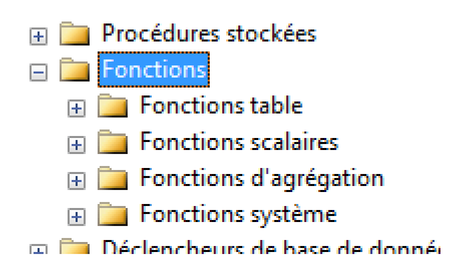
Le champ d'action d'une fonction est vraiment limité puisqu'il n'est possible de modifier que des objets locaux à la fonction. Il n'est pas possible de modifier des objets de la base, contenus à l'extérieur de la fonction. La création d'une fonction se fait par l'instruction DDL **CREATE FUNCTION**, habituelle lors de la création d'objets dans la base. De plus, il existe une multitude de fonctions système, utilisable par simple appel de celle-ci. Elles sont disponibles dans l'explorateur d'objets ainsi.



Détaillons maintenant tous les types de fonctions existants.

Création d'une fonction

La création d'une fonction se fait quasiment de la même manière pour les trois types. Nous allons bien sûr présenter les trois avec trois exemples, simplement nous allons le faire dans la même partie. Voici les deux syntaxes principales de création de fonctions (scalaire et table). Nous pouvons y accéder en déployant les nœuds de l'explorateur d'objet jusqu'au nœud « **Fonctions** », appliquer un clic droit sur le type de fonction à créer et choisir l'option « **Nouvelle fonction…** ».



Fonction scalaire

Une fonction scalaire est identique à une fonction mathématique : elle peut avoir plusieurs paramètres d'entrée (jusqu'à 1024), et renvoie une seule valeur.

```
CREATE FUNCTION <Scalar_Function_Name, sysname, FunctionName>
(
-- Add the parameters for the function here
<@Param1, sysname, @p1> <Data_Type_For_Param1, , int>
)
RETURNS <Function_Data_Type, ,int>
AS
BEGIN
-- Declare the return variable here
DECLARE <@ResultVar, sysname, @Result> <Function_Data_Type, ,int>
-- Add the T-SQL statements to compute the return value here
SELECT <@ResultVar, sysname, @Result> = <@Param1, sysname, @p1>
-- Return the result of the function
RETURN <@ResultVar, sysname, @Result>
```

```
END
GO
```

Ici, nous créons une fonction scalaire que l'on reconnaît grâce à l'instruction **RETURNS** int. On se sert bien entendu de l'instruction **CREATE FUNCTION** suivie du nom de la fonction à utiliser et des paramètres à prendre en compte entre parenthèses. Les clauses **AS BEGIN** et **END** déclarent dans l'ordre, le début et la fin de la fonction. Cette fonction retourne une valeur qui désigne le nombre d'articles présents, pour un stock passé en paramètre.

Dans les fonctions, certaines options sont disponibles, comme dans les procédures stockées. Elles vont vous permettre certaines actions sur cette fonction, et celle-ci est appelable directement après la définition du type de retour de la fonction :

- **WITH SCHEMABINDING** : Cette option permet de lier la fonction à tous les objets de la base auxquels elle fait référence. Dans ce cas, la suppression et la mise à jour de n'importe quel objet de la base, lié à la fonction est impossible.

- **WITH ENCRYPTION** : Permet de crypter le code dans la table système.

<u>Exemple 1</u> : Créons la fonction fn_DateFormat à partir de notre base de données « Papyrus ». La fonction fn_DateFormat formate à l'aide d'un séparateur entré en paramètre, une date et la retourne sous forme de chaîne de caractères.

```
CREATE FUNCTION vente.fn_DateFormat
(@pdate datetime, @psep char (1))
RETURNS char (10)
AS
BEGIN
RETURN
CONVERT (varchar(2), datepart (dd,@pdate))
+ @psep + CONVERT (varchar (2), datepart (mm, @pdate))
+ @psep + CONVERT (varchar (4), datepart (yyyy, @pdate))
END
```

Nous pouvons maintenant effectuer une requête pour connaître quelles sont les commandes passées du mois de mars et d'avril.

```
-- Quelles sont les commandes passées au mois de mars et d'avril ?
SELECT NUMCOM,vente.fn_DateFormat(DATCOM,'/') AS 'Date'
FROM vente.ENTCOM
WHERE MONTH(DATCOM) IN (3,4)
```

<u>Exemple 2</u> : Créons la function fn_NbCommandes qui renvoie le nombre de commandes pour un fournisseur donné avec une réponse formatée :

- Si le nombre de commandes est égal à 0, vous renverrez le message 'Aucune commande pour le founisseur xxxx'.

- Si le nombre de commandes est supérieur à 0, vous renverrez le message 'nnn commandes pour le fournisseur xxxx'.

```
CREATE FUNCTION vente.fn_NbCommandes
(@numfou int)
RETURNS varchar(50)
AS
BEGIN
DECLARE @nb int
DECLARE @com varchar(50)
SET @nb = (SELECT count(*) FROM vente.entcom
```

```
WHERE numfou = @numfou)
-- Si le nombre de commandes est égal à 0, vous renverrez le message
'Aucune commande pour le fournisseur xxxx'
IF @nb = 0
SET @com = 'Aucune commande pour le fournisseur ' + CAST(@numfou AS
varchar)
-- Si le nombre de commandes est supérieur à 0, vous renverrez le message
'nnn commandes pour le fournisseur xxxx'
ELSE
SET @com = CAST(@nb AS varchar) + ' commandes pour le fournisseur ' +
CAST(@numfou AS varchar)
RETURN @com
END
```

Nous pouvons maintenant tester notre fonction

```
SELECT DISTINCT vente.fn_NbCommandes(120) AS 'Nombre De Commandes'
      FROM vente.entcom
```

Exemple 3 : Créer la fonction scalaire fn_Date, qui avec l'indice de satisfaction en entrée, affiche un niveau de satisfaction en clair :

- Indice = Null, 'sans commentaire'
- Indice = 1 ou 2, 'Mauvais'
- Indice = 3 ou 4, 'Passable'
- Indice = 5 ou 6, 'Moyen'
- Indice = 7 ou 8, 'Bon'
- Indice = 9 ou 10, 'Excellent'

```
CREATE FUNCTION vente.fn_Date
(@indice int)
RETURNS varchar(50)
AS
BEGIN
DECLARE @com varchar(50)
SET @com = CASE
    WHEN @indice IS NULL THEN 'Sans commentaire'
      WHEN @indice > 0 and @indice < 3 THEN 'Mauvais'
      WHEN @indice > 2 and @indice < 5 THEN 'Passable'
      WHEN @indice > 4 and @indice < 7 THEN 'Moyen'
      WHEN @indice > 6 and @indice < 9 THEN 'Bon'
      WHEN @indice > 8 and @indice < 11 THEN 'Excellent'
      ELSE 'Hors limite'
    END
    RETURN @com
END
```

Testons notre fonction vente.fn_Date :

```
SELECT NOMFOU, vente.fn_Date(satisf) AS 'Satisfaction'
FROM vente.FOURNISSEUR
```

Fonction table

Les fonctions table sont de deux types :

- Les fonctions table en ligne, qui ne contiennent qu'une seule instruction **SELECT** déterminant le format de la table renvoyée.

- Les fonctions table multi instructions, qui déclarent le format d'une table virtuelle, avant de la remplir par des instructions **SELECT.**

```
CREATE FUNCTION <Inline_Function_Name, sysname, FunctionName>
(
-- Add the parameters for the function here
<@param1, sysname, @p1> <Data_Type_For_Param1, , int>,
<@param2, sysname, @p2> <Data_Type_For_Param2, , char>
)
RETURNS TABLE
AS
RETURN
(
-- Add the SELECT statement with parameter references here
SELECT 0
)
GO
```

Les fonctions table et table multi-instructions sont différentes dans le sens où le format de retour est différent. En effet, la fonction table retourne la solution d'un requête **SELECT**, alors que la fonction table multi-instruction, retournera une variable de type table, contenant l'instruction **SELECT** opérée par la fonction.

➔ Fonction table simple

```
CREATE FUNCTION recommander_stock (@Id int, @seuil int)
RETURNS TABLE
AS
RETURN (SELECT * FROM Stock WHERE Id_Stock = @Id AND Quantite < @Seuil)
```

Dans ce cas-là, après l'instruction de création de fonction, **CREATE FUNCTION**, on indique simplement que la fonction retourne une donnée de type table avec la clause **RETURNS TABLE**. Par la suite, après la clause **AS**, on précise quelle instruction doit être retournée dans la valeur de retour de type table de la fonction.

Exemple 1 : Créer la fonction fn_CA_Fournisseur, qui en fonction d'un code fournisseur et d'une année entrés en paramètre, restituera le CA potentiel de ce fournisseur pour l'année souhaitée.

```
CREATE FUNCTION vente.fn_CA_Fournisseur
(@codfou int, @datcom date)
RETURNS TABLE
AS
RETURN (
SELECT f.NUMFOU, NOMFOU, CAST(sum(QTECDE * (CAST(priuni as money) *
1.2060)) as money) as 'CA'
FROM vente.FOURNISSEUR f
INNER JOIN vente.ENTCOM e
ON f.NUMFOU = e.NUMFOU
INNER JOIN vente.ligcom
ON e.NUMCOM = vente.ligcom.NUMCOM
WHERE e.NUMFOU = @codfou
AND YEAR (@datcom) = YEAR(GETDATE())
GROUP BY f.NUMFOU, NOMFOU)
```

Testons notre fonction :

```
SELECT * FROM vente.fn_CA_Fournisseur('120', '27/01/2011')
```

<u>Exemple 2</u> : Utilisons l'option **SCHEMABINDING**. **SCHEMABINDING** indique que la fonction est liée aux objets de base de données auxquels elle fait référence. Toutes modification (**ALTER**) ou suppression (**DROP**) de ces objets sont vouées à l'échec. La liaison de la fonction aux objets auxquels elle fait référence est supprimée uniquement lorsqu'une des actions suivantes se produit :

- La fonction est supprimée.
- La fonction est modifiée, avec l'instruction **ALTER**, sans spécification de l'option **SCHEMABINDING**.

Créez une table FOURNIS_IND de structure identique à FOURNISSEUR de notre base de données « Papyrus » et insérez les lignes de la table FOURNISSEUR dont la colonne indice de satisfaction est 'Bon'.

```
CREATE FUNCTION vente.fournis
(@com varchar(50))
RETURNS TABLE
AS
RETURN (
SELECT *
FROM vente.FOURNISSEUR
-- Nous utilisons une fonction créée précédemment
WHERE vente.fn_Date(satisf) = @com)
```

Testons notre fonction vente.fn_CA_Fournisseur :

```
SELECT * FROM vente.fournis('Bon')
```

On créer la table FOURNIS_IND.

```
CREATE TABLE FOURNIS_IND (NUMFOU int, NOMFOU varchar(30), RUEFOU
varchar(30), POSFOU varchar(5), VILFOU varchar(30), CONFOU varchar(15),
SATISF tinyint)
```

On insère les données :

```
INSERT INTO FOURNIS_IND (NUMFOU, NOMFOU, RUEFOU, POSFOU, VILFOU, CONFOU,
SATISF)
SELECT *
FROM vente.fournis('Bon')
```

On modifie le schéma :

```
ALTER SCHEMA vente TRANSFER dbo.FOURNIS_IND
```

Sans ce schéma, il sera impossible de définir l'option **WITH SCHEMABINDING** pour l'énoncer ci-dessous.

On créez la fonction fn_ Compte, avec l'option **SCHEMABINDING** qui délivre le nombre de lignes de la table FOURNIS_IND.

```
CREATE FUNCTION vente.fn_Compte
(@com varchar(50))
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN (
SELECT count(*) AS 'Nbr Lignes'
```

```
FROM vente.FOURNIS_IND
)
```

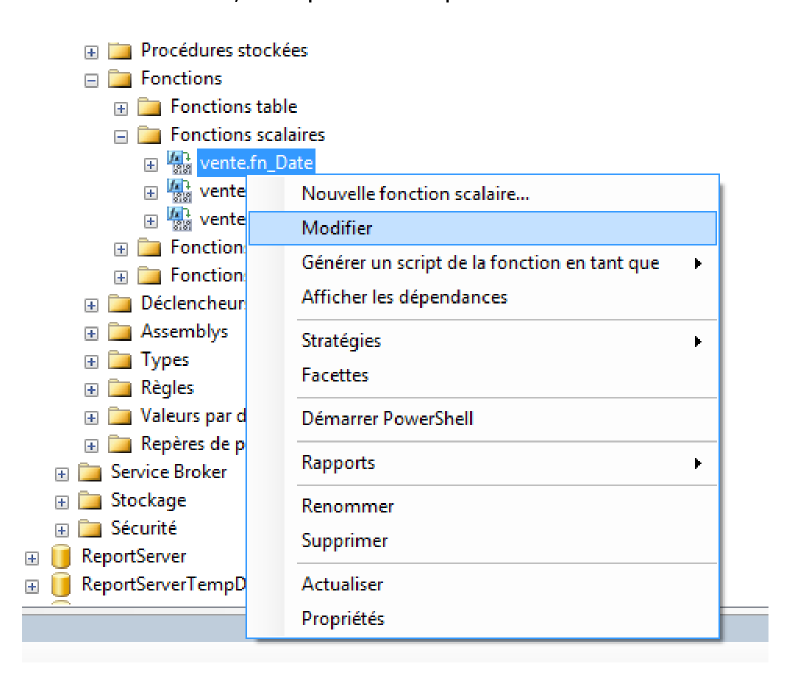
➔ Fonction table multi-instruction

```
CREATE FUNCTION table_multi (@Id int)
RETURNS @variable TABLE (Id_Stock int, Quantite int, Nom_Entrepos
varchar(25))
AS
BEGIN
SELECT @variable = (SELECT Id_Stock, Quantite, Nom_Entrepos
FROM Entrepos E INNER JOIN Stock S
ON E.Id_Entrepos = S.Id_Entrepos
WHERE S.Id_Stock = @Id)
RETURN
END
```

Dans ce cas, en revanche, la valeur de retour est toujours une table, simplement, on donne à une variable ce type et on définit les colonnes qu'elle contient. Les valeurs définies par le **SELECT** y seront alors contenues. Il faut faire attention, en contrepartie, à ce que le nombre de colonnes dans la variable retournée par la fonction ait le même nombre de colonnes que le résultat retourné par l'instruction **SELECT**.

Modification d'une fonction

La modification d'une fonction n'est possible que par une instruction T-SQL DDL, l'instruction **ALTER FUNCTION**. Par l'interface, lorsque l'on clique droit « **Modifier** » :



Nous arrivons directement sur notre code Transaq-SQL :

```
SQLQuery3.sql - (l...12-375\cdi01 (55))   SQLQuery1.sql - (...2-375\cdi01 (52))*   03112-375.
USE [Papyrus]
GO
/****** Object:  UserDefinedFunction [vente].[fn_Date]    Sc
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER FUNCTION [vente].[fn_Date]
(@indice int)
RETURNS varchar(50)
AS
BEGIN
DECLARE @com varchar(50)
SET @com = CASE
    WHEN @indice IS NULL THEN 'Sans commentaire'
    WHEN @indice > 0 and @indice < 3 THEN 'Mauvais'
    WHEN @indice > 2 and @indice < 5 THEN 'Passable'
    WHEN @indice > 4 and @indice < 7 THEN 'Moyen'
    WHEN @indice > 6 and @indice < 9 THEN 'Bon'
    WHEN @indice > 8 and @indice < 11 THEN 'Excellent'
    ELSE 'Hors limite'
    END
    RETURN @com
END
```

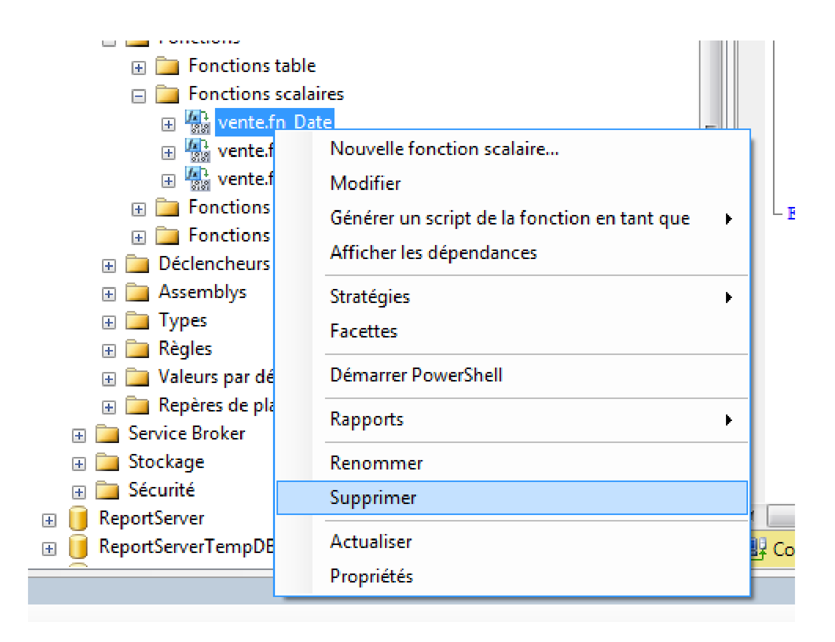
En effet, dans ce cas-là, les commandes **CREATE FUNCTION** et **ALTER FUNTION** auront quasiment la même fonction, puisque dans les deux cas, tout le corps de la fonction doit être réécrit et totalité. Il n'est pas possible d'ajouter ou supprimer seulement une seule ligne dans celui-ci. Leur seule différence réside donc dans le fait que la fonction soit créée ou modifiée. Voyons donc la syntaxe qui permet la modification d'une fonction en Transact SQL :

```
ALTER FUNCTION nombre_element_stock (@Entrepos int)
RETURNS int
AS
BEGIN
DECLARE @nb int
SELECT @nb = COUNT(Id_Stock)
FROM Stock
WHERE Id_Entrepos = @Entrepos
RETURN @nb
END
GO
```

On remarque donc aisément que la seule modification qu'il existe entre la création et la modification d'une fonction réside dans le remplacement du mot clé **CREATE FUNCTION** par le mot clé **ALTER FUNCTION**.

Suppression d'une fonction

La suppression d'une fonction peut, comme pour les procédures stockées ou tout autre objet de la base, être faite de deux manières. La première est la méthode graphique. Nous ne détaillerons pas cette méthode, puisqu'elle est la même pour tout objet de la base. Nous rappellerons juste qu'il vous suffit d'étendre, dans votre explorateur d'objet, les nœuds correspondants au chemin de votre fonction, de faire un clic droit sur celle-ci, et de sélectionner « **Supprimer** ».



Avec le langage Transact SQL, la méthode est aussi la même que pour tout autre objet de la base. Nous utiliserons l'instruction DROP et nous l'adapterons au cas d'une fonction. Voici la méthode type de suppression d'une fonction par code T-SQL :

```
DROP FUNCTION nombre_element_stock
```

L'instruction **DROPFUNCTION** nous permet donc de supprimer une fonction, de n'importe quel type. Il suffit juste de préciser le nom de la fonction après cette instruction.

Procédures Stockées

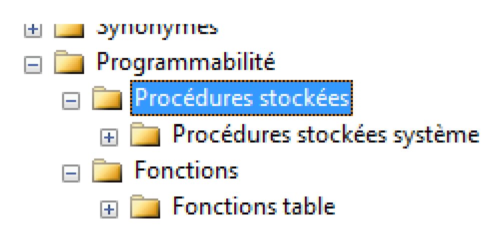
Les procédures stockées sont des ensembles d'instructions du DML, pouvant être exécutées par simple appel de leur nom ou par l'instruction **EXECUTE**. Les procédures stockées sont de véritables programmes qui peuvent recevoir des paramètres, être exécutés à distance, renvoyer des valeurs et possédant leurs propres droits d'accès (**EXECUTE**). Celles-ci sont compilées une première fois, puis placées en cache mémoire, ce qui rend leur exécution plus performante du fait que le code soit précompilé. Les procédures stockées sont contenues dans la base de données, et sont appelables par leurs noms. Il existe une multitude de procédures stockées pré intégré dans SQL Server lors de l'installation qui servent principalement à la maintenance des bases de données utilisateur. Celle-ci commence toujours par les trois caractères « sp_ » comme **stored procedure**. Pour résumer les avantages des procédures stockées, nous allons lister leurs utilisations :

- Accroissement des performances.
- Sécurité d'exécution.
- Possibilité de manipuler les données système.
- Implémente le traitement en cascade et l'enchaînement d'instructions.

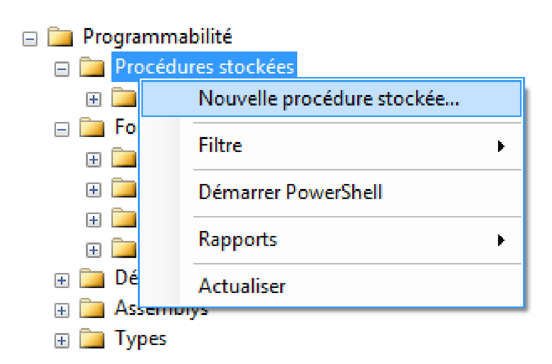
Création d'une procédure stockée

Pour créer une procédure stockée, nous sommes obligés de passer par du code T-SQL. En revanche, il existe un assistant de génération automatique de la structure d'une procédure stockée. Nous allons tout d'abord étudier la structure générale d'une procédure stockée avec cette génération automatique, puis nous donnerons un exemple, présent dans le script de la base que nous utilisons, pour bien comprendre les notions exposées sur les procédures stockées.

Tout d'abord, pour générer le script automatiquement, étendez les nœuds de l'explorateur d'objet comme ceci :



Maintenant, pour créer une nouvelle procédure stockée en générant le code automatiquement, il vous suffit de faire un clic droit sur le nœud « **Procédures stockées** » et de choisir l'option « **Nouvelle procédure stockée…** ».



Une nouvelle fenêtre de requête s'ouvre dans SSMS, vous proposant le code pour créer une nouvelle procédure stockée. Le code est le suivant :

```
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
-- Add the parameters for the stored procedure here
<@Param1, sysname, @p1> <Datatype_For_Param1, , int> =
<Default_Value_For_Param1, , 0>,
<@Param2, sysname, @p2> <Datatype_For_Param2, , int> =
<Default_Value_For_Param2, , 0>
AS
BEGIN
-- SET NOCOUNT ON added to prevent extra result sets from
-- interfering with SELECT statements.
SET NOCOUNT ON;
-- Insert statements for procedure here
SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

Nous allons maintenant détailler le code.

```
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
```

Nous créons une procédure stockée avec l'instruction DDL **CREATE PROCEDURE** suivie du nom à donner à la procédure. Ce nom vous permettra de l'appeler et de la reconnaître dans la base.

```
-- Add the parameters for the stored procedure here
```

```
<@Param1, sysname, @p1> <Datatype_For_Param1, , int> =
<Default_Value_For_Param1, , 0>,
<@Param2, sysname, @p2> <Datatype_For_Param2, , int> =
<Default_Value_For_Param2, , 0>
```

Nous devons ensuite préciser les variables que prend en paramètre la procédure stockée, durant son appel. Ces variables vont nous servir par la suite dans la définition des actions que la procédure stockée fait. Nous pouvons initialiser ou non les variables, le plus important est bien entendu de donner un nom conventionnel et un type de donnée à nos variables.

```
AS
BEGIN
-- SET NOCOUNT ON added to prevent extra result sets from
-- interfering with SELECT statements.
SET NOCOUNT ON;
-- Insert statements for procedure here
SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

Les instructions **AS BEGIN** et **END** sont les délimiteurs du code à utiliser par la procédure stockée. Toutes les instructions comprises entre ces deux mots clés seront prises en compte et exécutées par la procédure stockée.

Maintenant que nous avons présenté la structure générale de création d'une procédure stockée, prenons des exemples concrets pour l'illustrer à partir de notre base de données « Papyrus ».

Exemple 1 : Créez la procédure stockée **Lst_fournis** correspondante à la requête : « Afficher le code des fournisseurs pour lesquels une commande a été passée ».

```
USE Papyrus
GO
CREATE PROCEDURE Lst_fournis2
AS
SELECT DISTINCT NUMFOU
FROM vente.ENTCOM
```

On exécute notre procédure stockée pour la vérifier :

```
EXEC Lst_fournis
```

EXEC ou **EXECUTE** est une instruction permettant de lancer une requête ou une procédure stockée au sein d'une procédure ou un trigger. La plupart du temps il n'est pas nécessaire d'utilise l'instruction **EXEC**, si l'intégralité de la commande SQL ou de la procédure à lancer est connue. Mais lorsqu'il s'agit par exemple d'un ordre SQL contenant de nombreux paramètres, alors il est nécessaire de le définir dynamiquement.

	NUMFOU
1	120
2	540
3	8700
4	9150
5	9180

Après la déclaration de création de procédures, et la déclaration des paramètres, il est possible de définir des options grâce à une clause **WITH** ou une clause **FOR**. Ces options sont les suivantes :

- **WITH RECOMPILE** : la procédure sera recompilée à chaque exécution.

- **WITH ENCRYPTION** : Permet de crypter le code dans la table système.

- **FOR REPLICATION** : Permet de préciser que la procédure sera utilisée lors de la réplication.

Exemple 2 : Reprenons l'exemple précèdent avec l'option **WITH RECOMPILE**.

```
CREATE PROCEDURE Lst_fournisBis
WITH RECOMPILE
AS
SELECT DISTINCT NUMFOU
FROM vente.ENTCOM
```

On teste notre procédure stockée de la même façon :

```
EXEC Lst_fournis
```

SQL Server recompile automatiquement les procédures stockées et les déclencheurs quand il est avantageux de le faire.

Cependant, vous pouvez forcer la recompilation des procédures stockées et des déclencheurs lors de leur prochaine exécution. L'exemple suivant engendre la recompilation des procédures stockées et des déclencheurs qui agissent sur la table PRODUIT lors de leur prochaine exécution.

```
USE Papyrus;
GO
EXEC sp_recompile N'vente.PRODUIT';
GO
```

```
Messages
L'objet 'vente.PRODUIT' a été marqué correctement pour la recompilation.
```

Exemple 3 : Création d'une procédure stockée avec un paramètre en entrée. On créer la procédure stockée « Lst_Commandes », qui liste les commandes ayant un libellé particulier dans le champ OBSCOM.

```
CREATE PROCEDURE Lst_Commandes
@libelle varchar(50)
AS
SELECT e.NUMCOM, NOMFOU, LIBART, QTECDE * cast(PRIUNI as money) as 'Sous
Total'
FROM vente.ENTCOM e
INNER JOIN vente.FOURNISSEUR f
ON e.NUMFOU = f.NUMFOU
INNER JOIN vente.LIGCOM l
ON e.NUMCOM = l.NUMCOM
INNER JOIN vente.PRODUIT p
```

```
ON l.CODART = p.CODART
WHERE e.OBSCOM like (@libelle)
```

On teste notre procédure stockée : Il est alors nécessaire d'entrer des valeurs pour les paramètres indiqués dans la procédure stockée.

```
EXEC Lst_Commandes '%urgent%'
```

	NUMCOM	NOMFOU	LIBART	Sous Total
1	9	ECLIPSE	Papier 1 ex continu	141000,00
2	9	ECLIPSE	Papier 2 ex continu	97000,00
3	9	ECLIPSE	Papier 3 ex continu	68000,00
4	9	ECLIPSE	CD R slim 80 mm	4000,00
5	9	ECLIPSE	Pré imprimé commande	2100000,00
6	9	ECLIPSE	Pré imprimé bulletin paie	1200000,00
7	11	DEPANPAP	Papier 1 ex continu	59000,00
8	11	DEPANPAP	Papier 2 ex continu	29500,00

Exemple 4 : Création d'une procédure stockée avec des paramètres en entrée et en sortie. On créer la procédure stockée « CA_Fournisseur », qui pour un code fournisseur et une année entré en paramètre, calcule et restitue le CA potentiel de ce fournisseur pour l'année souhaitée. On exécutera la requête que si le code fournisseur est valide, c'est-à-dire qui existe dans la table FOURNISSEUR, sinon on renverra un code d'état égal à −100.

```
CREATE PROCEDURE CA_Fournisseur
@numfou int,
@datcom date
AS
BEGIN
/* Empêche le message indiquant le nombre de lignes concernées par une
instruction
ou une procédure stockée Transact-SQL d'être renvoyé avec l'ensemble de
résultats. */
SET NOCOUNT ON;
-- On déclare nos variables
DECLARE @comm varchar(50)
-- Si le fournisseur existe
IF EXISTS(SELECT NOMFOU FROM vente.FOURNISSEUR WHERE NUMFOU = @numfou)
BEGIN
SET @comm = 'FOURNISSEUR : '+ Convert(varchar(25),@numfou) + ' ' + @comm
PRINT @comm
PRINT 'LISTE DES ARTICLES EN COMMANDE'
/* La requête suivante calcule le chiffre d'affaires par fournisseur pour
l'année donnée
sachant que les prix indiqués sont hors tars et que le taux TVA est de
20,60 % */
SELECT f.NUMFOU, NOMFOU, CAST(sum(QTECDE * (CAST(priuni as money) *
1.2060)) as money) as 'CA'
FROM vente.FOURNISSEUR f
INNER JOIN vente.ENTCOM e
ON f.NUMFOU = e.NUMFOU
INNER JOIN vente.ligcom
ON e.NUMCOM = vente.ligcom.NUMCOM
WHERE e.NUMFOU = @numfou
AND YEAR (DATCOM) = YEAR(GETDATE())
GROUP BY f.NUMFOU, NOMFOU
```

```
-- Si le fournisseur exist, on retourne le code 0 après avoir exécuté la
requête
RETURN 0;
END
-- Si le fournisseur n'existe pas, on retourne le code -100
ELSE
RETURN -100;
END
GO
```

On teste notre procédure stockée :

```
USE Papyrus
DECLARE @retour int
DECLARE @date date
DECLARE @lefourn int
SET @lefourn = 120
SET @date = '27/01/2011'
EXEC @retour = CA_Fournisseur @lefourn, @date
IF @retour = 0
PRINT 'Le fournisseur ' + convert(varchar(4), @lefourn) + ' existe bien !'
ELSE
PRINT 'Le fournisseur ' + convert(varchar(4), @lefourn) + ' n''existe pas
!'
```

	NUMFOU	NOMFOU	CA
1	120	GROBRIGAN	14188590,00

Sur l'onglet « **Messages** » nous obtenons l'information suivante :

Résultats | Messages

```
LISTE DES ARTICLES EN COMMANDE
Le fournisseur 120 existe bien !
```

```
USE Papyrus
DECLARE @retour int
DECLARE @date date
DECLARE @lefourn int
SET @lefourn = 100
SET @date = '27/01/2011'
EXEC @retour = CA_Fournisseur @lefourn, @date
IF @retour = 0
PRINT 'Le fournisseur ' + convert(varchar(4), @lefourn) + ' existe bien !'
ELSE
PRINT 'Le fournisseur ' + convert(varchar(4), @lefourn) + ' n''existe pas
!'
```

Messages

```
Le fournisseur 100 n'existe pas !
```

Exemple 5 : Créez une nouvelle procédure stockée « CA2_ Fournisseur » qui lorsque le code FOURNISSEUR n'existe pas, renvoie le message s'inscrivant dans le journal d'erreurs du serveur et dans le journal des évènements.

Avant toute chose, nous devons créer le message utilisateur : « Fournisseur inexistant » grâce à la procédure stockée **sp_addmessage**.

```
USE Papyrus
EXECUTE sp_addmessage 50001, 16, 'Fournisseur inexistant', 'us_english'
EXECUTE sp_addmessage 50001, 16, 'Fournisseur inexistant'
GO
```

Tous les messages stockés à l'aide de **sp_addmessage** peuvent être affichés grâce à l'affichage catalogue sys.messages.

```
SELECT * FROM SYS.MESSAGES
```

Ensuite, nous créons notre procédure : on peut déclarer des valeurs par défaut et spécifier si le paramètre est en sortie avec le mot clef **OUTPUT**.

```
CREATE PROCEDURE CA2_Fournisseur
@numfou int,
@datcom date,
@nbrlign int output
AS
BEGIN
/* Empêche le message indiquant le nombre de lignes concernées par une
instruction
ou une procédure stockée Transact-SQL d'être renvoyé avec l'ensemble de
résultats. */
SET NOCOUNT ON;
-- On déclare nos variables
DECLARE @comm varchar(50)
-- Si le fournisseur existe
      -- on insère une clause try
      BEGIN TRY
            IF EXISTS(SELECT NOMFOU FROM vente.FOURNISSEUR WHERE NUMFOU =
@numfou)
                  BEGIN
                        SET @comm = 'FOURNISSEUR : '+
Convert(varchar(25),@numfou) + ' ' + @comm
                        PRINT @comm
                        PRINT 'LISTE DES ARTICLES EN COMMANDE'
                        /* La requête suivante calcule le chiffre
d'affaires par fournisseur pour l'année donnée
                        sachant que les prix indiqués sont hors tars et que
le taux TVA est de 20,60 % */
                        SELECT f.NUMFOU, NOMFOU, CAST(sum(QTECDE *
(CAST(priuni as money) * 1.2060)) as money) as 'CA'
                        FROM vente.FOURNISSEUR f
                        INNER JOIN vente.ENTCOM e
                        ON f.NUMFOU = e.NUMFOU
                        INNER JOIN vente.ligcom
                        ON e.NUMCOM = vente.ligcom.NUMCOM
                        WHERE e.NUMFOU = @numfou
                        AND YEAR (@datcom) = YEAR(GETDATE())
                        GROUP BY f.NUMFOU, NOMFOU
                        SET @nbrlign = (SELECT count(*)
                        FROM vente.ENTCOM e
```

```
                        WHERE e.NUMFOU = @numfou)
                        -- Si le fournisseur existe, on retourne le code 0
après avoir exécuté la requête
                        RETURN 0;
                  END
            ELSE
                  BEGIN
                        RAISERROR(50001 , 16, 1, @numfou) WITH LOG
                        -- Si le fournisseur n'existe pas, on retourne le
code -100
                        RETURN -100;
                  END
      END TRY
      BEGIN CATCH
            -- Erreur traitée dans le bloc catch
            PRINT 'Erreur intercepté ici !'
      END CATCH
END
GO
```

On test la procédure : Pour récupérer la valeur d'un paramètre **OUTPUT** il faut préciser une variable de récupération de nature **OUTPUT** dans le lancement de l'exécution.

```
USE Papyrus
DECLARE @retour int
DECLARE @datecom date
DECLARE @founisseur int
DECLARE @lignes int
SET @founisseur = 125450
SET @datecom = '27/01/2011'
SET @lignes = 0
BEGIN TRY
      EXEC @retour = CA2_Fournisseur @founisseur, @datecom, @lignes output

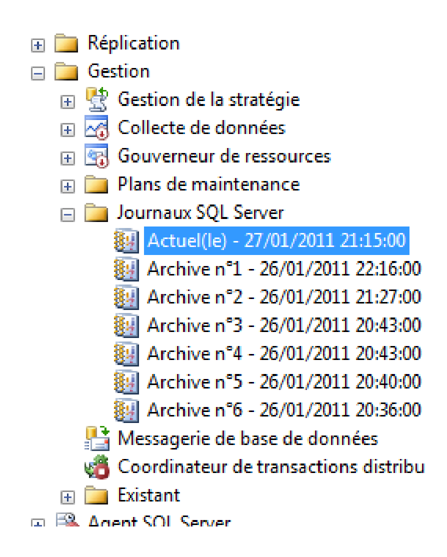
      IF @retour = 0 And @lignes > 0
      PRINT 'Le fournisseur ' + convert(varchar(4), @founisseur) + ' existe
bien !'
      + ' Il a ' + convert(varchar(5), @lignes) + ' commandes le
concernant.'
      ELSE IF @retour = 0 And @lignes < 0
      PRINT 'Le fournisseur ' + convert(varchar(4), @founisseur) + ' existe
bien '
      + 'mais il n''a pas de commandes !'
      ELSE
      PRINT 'Le fournisseur ' + convert(varchar(4), @founisseur) + '
n''existe pas !'
END TRY
BEGIN CATCH
-- Instruction si erreur
PRINT convert(varchar(5), ERROR_NUMBER()) + ' ' + ERROR_MESSAGE()
END CATCH
```

```
Messages
Erreur intercepté ici !
Le fournisseur * n'existe pas !
```

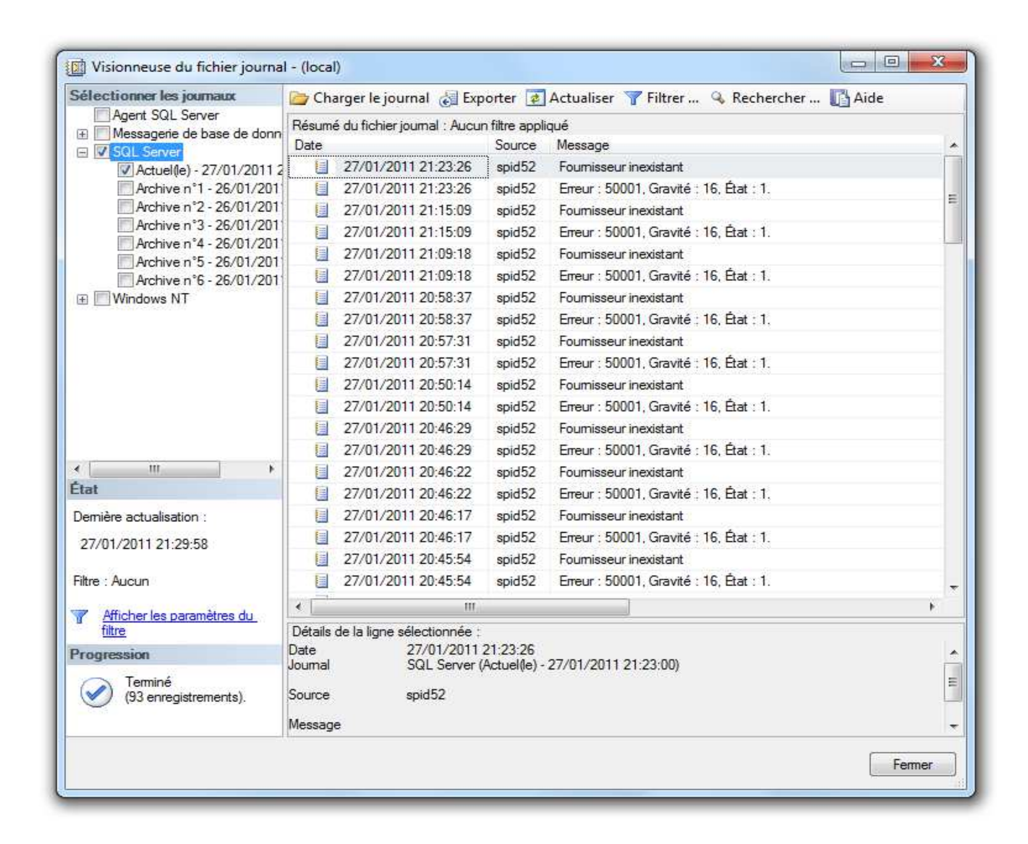
Nous allons maintenant consulter le journal des erreurs de SQL Server.

Consultez le journal des erreurs de SQL Server pour vérifier que l'exécution des processus a réussi (par exemple les opérations de sauvegarde et de restauration, les commandes de traitement ou d'autres scripts et processus). Cela peut s'avérer utile pour détecter tout problème en cours ou potentiel, y compris les messages de récupération automatique (surtout si une instance de SQL Server a été arrêtée puis redémarrée), les messages du noyau ou autre message d'erreur de niveau serveur.

Consultez le journal des erreurs SQL Server à l'aide de SQL Server Management Studio ou de n'importe quel éditeur de texte. Dans « SQL Server Management Studio », ouvrez le dossier « **Gestion** » puis « **Journaux SQL Server** ».



Cliquez sur le dernier en date :



Par défaut, le journal des erreurs se trouve dans les fichiers « Program Files\Microsoft SQL Server\MSSQL.n\MSSQL\LOG\ERRORLOG » et **ERRORLOG.n**.

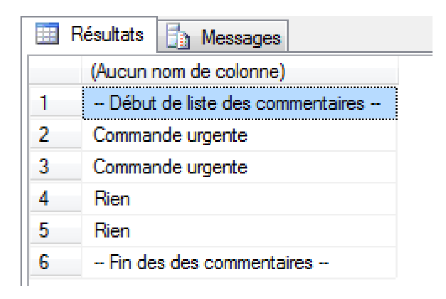
Un nouveau journal des erreurs est créé à chaque démarrage d'une instance de SQL Server, bien que la procédure stockée système **sp_cycle_errorlog** puisse être utilisée pour recycler les fichiers du journal des erreurs sans devoir redémarrer l'instance de SQL Server. En principe, SQL Server conserve une copie de sauvegarde des six derniers journaux des erreurs, attribuant l'extension .1 au plus récent, l'extension .2 à celui qui le précède, et ainsi de suite. Le journal des erreurs en cours n'a aucune extension.

<u>Exemple 6</u> : On recherche toutes les lignes de commentaires pour les fournisseurs se situant entre telle et telle fourchette.

```
CREATE PROCEDURE liste_com
@com_debut INTEGER, @com_fin INTEGER
AS
SELECT '-- Début de liste des commentaires --'
UNION ALL
SELECT OBSCOM
FROM vente.ENTCOM
WHERE NUMFOU BETWEEN @com_debut AND @com_fin
UNION ALL
SELECT '-- Fin des commentaires --'
```

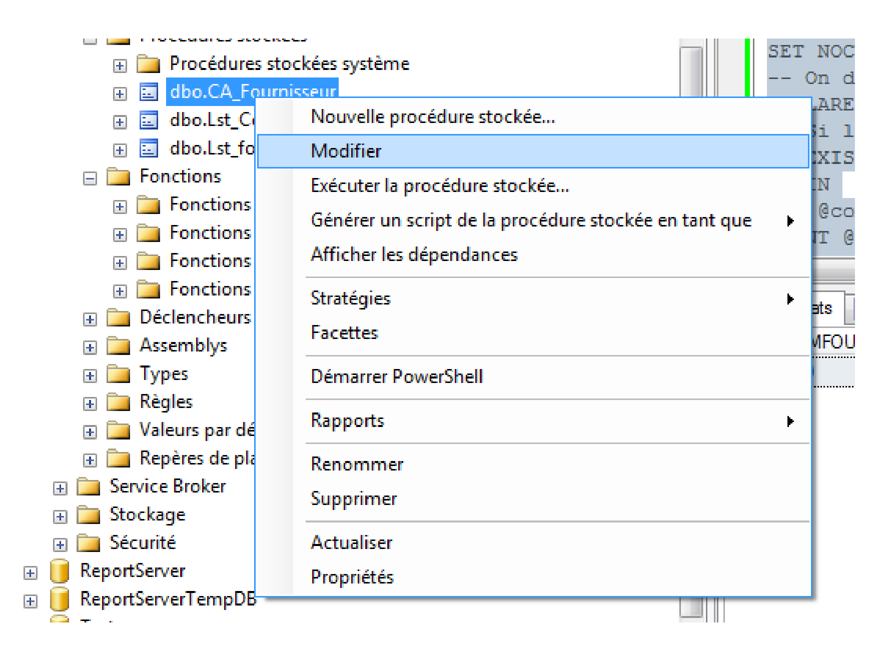
On exécute pour les fournisseurs de 9000 à 10000 :

```
EXEC liste_com '9000', '10000'
```



Modifier une procédure stockée

La modification d'une procédure stockée ne peut se faire que par l'interface graphique en premier lieu. Pour en modifier une, étendez tous les nœuds qui mènent à une procédure stockée en particulier, comme ceci :



Pour modifier une procédure stockée en particulier, il vous suffit alors de procéder à un clic droit sur celle-ci, et de choisir l'option « **Modifier** ». Une nouvelle fenêtre de requête apparaît, avec le code que vous aviez entré lors de la première création de votre procédure. Modifiez-le comme voulu et ré exécutez le pour modifier la procédure.

Suppression d'une procédure stockée

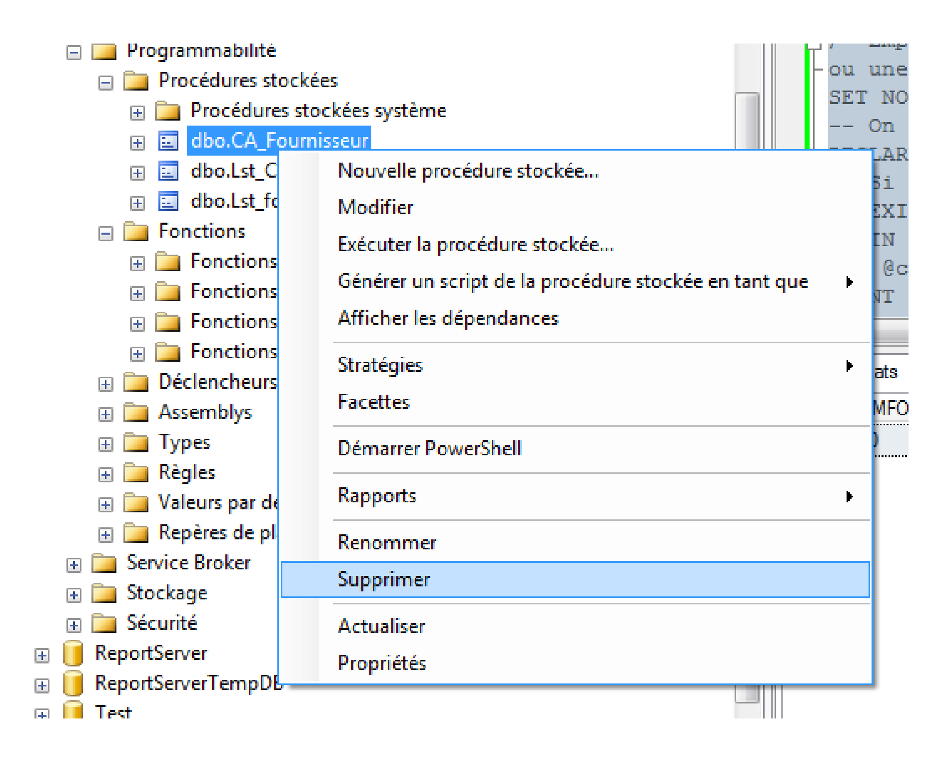
La suppression d'une procédure peut se faire en revanche de deux manières, avec du code ou l'interface graphique. Nous allons présenter les deux.

- Avec du code, il est très simple de supprimer une procédure stockée. Regardons la syntaxe générale :

```
USE Entreprise
DROP PROCEDURE Ajout_Client
```

Dans un premier temps, avec l'instruction **USE**, placez-vous dans la base contenant la procédure stockée à supprimer. Si vous ne le faites pas, la procédure stockée sera indiquée comme un élément non existant dans la base par SQL Server. Utilisez ensuite les mots clé **DROP PROCEDURE** suivit du nom de la procédure stockée pour supprimer celle-ci. Exécutez le code. Vous venez de supprimer votre procédure stockée.

- Avec l'interface graphique, nous allons procéder comme pour une modification de la procédure. Étendez dans un premier temps tous les nœuds qui mènent à la procédure stockée à supprimer, comme ceci :



Opérez alors un simple clic droit sur la procédure stockée choisie, et sélectionnez l'option « **Supprimer** ». Une nouvelle fenêtre apparaît. Il s'agit simplement d'une validation de votre choix, aucune autre option n'est disponible. Cliquez sur « **OK** ». Votre procédure stockée est supprimée.

Les curseurs

Le curseur est un mécanisme de mise en mémoire en tampon permettant de parcourir les lignes d'enregistrements du résultat renvoyé par une requête. Les curseurs sont envoyés par MS-SQL Server tout le temps, mais on ne voit pas le mécanisme se passer, ainsi lors d'une requête **SELECT**, SQL Server va employer des curseurs.

Pour utiliser un curseur, il faut commencer par le déclarer :

```
DECLARE name CURSOR FOR
      SELECT …
```

On peut aussi l'utiliser avec de la modification en ajoutant **FOR UPDATE** à la fin de la requête, bien que ce ne soit pas conseillé.

Ensuite, il faut ouvrir ce curseur avec **OPEN name** et ne pas oublier de le fermer à la fin avec **CLOSE name**. Il faut aussi utiliser **DEALLOCATE** pour libérer la mémoire du curseur.

Pour récupérer les valeurs actuelles contenues dans le curseur, il faut employer :

```
FETCH name INTO @value1, @value2 …
```

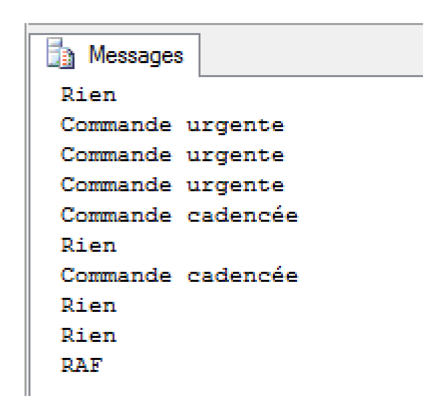
Cela va stocker les valeurs actuelles de l'enregistrement courant dans les variables **@valueX**, qu'il ne faut surtout pas oublier de déclarer.

On peut néanmoins utiliser **FETCH** pour d'autres choses :

• Aller à la première ligne : **FETCH FIRST FROM curseur_nom**
• Aller à la dernière ligne : **FETCH LAST FROM curseur_nom**
• Aller à la ligne suivante : **FETCH NEXT FROM curseur_nom**
• Aller à la ligne précédente : **FETCH PRIOR FROM curseur_nom**
• Aller à la ligne X : **FETCH ABSOLUTE ligne FROM curseur_nom**
• Aller à X lignes plus loin que l'actuelle : **FETCH RELATIVE ligne FROM curseur_nom**

Pour parcourir un curseur, on peut employer une boucle **WHILE** qui teste la valeur de la fonction **@@FETCH_STATUS** qui renvoie 0 tant que l'on n'est pas à la fin.

```
DECLARE @numfou VARCHAR(50)
DECLARE curseur_comparfou CURSOR FOR
SELECT OBSCOM FROM vente.ENTCOM
OPEN curseur_comparfou
FETCH curseur_comparfou INTO @numfou
WHILE @@FETCH_STATUS = 0
BEGIN
      PRINT @numfou
      FETCH curseur_comparfou INTO @numfou
END
CLOSE curseur_comparfou
DEALLOCATE curseur_comparfou
```



Ici, nous récupérons toutes les observations de la colonne OBSCOM de la table ENTCOM. Modifions l'exemple précèdent de façon à l'incorporé dans une procédure stockée et de nous donner les observations d'un fournisseur précis.

```
CREATE PROCEDURE recherche_ligne_com
-- On déclare nos variables
@numfou VARCHAR(50)
AS
BEGIN
-- On désigne curseur_comparfou
DECLARE curseur_comparfou CURSOR FOR
-- Le curseur agi pour la sélection suivante
SELECT OBSCOM FROM vente.ENTCOM WHERE NUMFOU = @numfou
-- On ouvre le curseur
OPEN curseur_comparfou
FETCH curseur_comparfou INTO @numfou
WHILE @@FETCH_STATUS = 0
BEGIN
      PRINT @numfou
      FETCH curseur_comparfou INTO @numfou
```

```
END
-- On ferme le curseur
CLOSE curseur_comparfou
-- On le vide de la mémoire
DEALLOCATE curseur_comparfou
END
GO
```

On interroge notre procédure stockée :

```
EXEC recherche_ligne_com3 '120'
```

```
Messages
Rien
Commande cadencée
RAF
```

On peut bien entendu imbriquer plusieurs curseurs les uns dans les autres pour des choses plus compliquées.

Concrètement, maintenant que nous avons vu comment fonctionnait un curseur et comment l'employer, que fait-il de plus qu'une simple requête ? Il permet surtout d'intervenir sur le résultat de la requête. On peut intervenir sur chaque valeur retournée, on peut modifier ces valeurs ou supprimer des lignes. On peut aussi réaliser des opérations avec ces données avant qu'elles arrivent au programme qui les utilise, c'est à dire des calculs de somme, des maximums, des modifications de date, des formatages de chaînes de caractères.

Un exemple intéressant est le parcours avec rupture, c'est à dire parcourir et si on a déjà eu une fois cet objet on ne le réaffiche pas. Dans l'exemple que je vais vous présenter, on affiche tous les genres, les acteurs par genre et pour chaque acteur les livres qu'ils ont écrits. On emploie des ruptures pour vérifier que l'on n'a pas déjà affiché une fois cet élément :

```
DECLARE @titre VARCHAR(50), @genre VARCHAR(50), @rupture_genre VARCHAR(50),
@rupture_auteur VARCHAR(50), @auteur VARCHAR(50)
DECLARE @i_genre INT
SET @i_genre = 1
SET @rupture_genre = ''
SET @rupture_auteur = ''

DECLARE curseur_ouvrages CURSOR FOR
      SELECT ouvrage_titre, genre_nom, auteur_nom FROM t_ouvrages O
      LEFT OUTER JOIN t_genres
            ON genre_id = ouvrage_genre
      LEFT OUTER JOIN t_ouvrages_auteurs TOA
            ON TOA.ouvrage_id = O.ouvrage_id
      LEFT OUTER JOIN t_auteurs TA
            ON TA.auteur_id = TOA.auteur_id
      ORDER BY ouvrage_genre

OPEN curseur_ouvrages

FETCH curseur_ouvrages INTO @titre , @genre, @auteur

WHILE @@FETCH_STATUS = 0
BEGIN
```

```
      IF @genre != @rupture_genre
      BEGIN
            PRINT ''
            PRINT CONVERT(CHAR(2),@i_genre) + '. ' + @genre

            SET @i_genre = @i_genre + 1
            SET @rupture_auteur = ''
      END

      IF @auteur != @rupture_auteur
      BEGIN
            PRINT ''
            PRINT @auteur
            PRINT '------------'
      END

      PRINT @titre;

      SET @rupture_genre = @genre
      SET @rupture_auteur = @auteur
      FETCH curseur_ouvrages INTO @titre , @genre, @auteur
END

CLOSE curseur_ouvrages
DEALLOCATE curseur_ouvrages
```

Fonctions de curseurs

Il y a trois fonctions intéressantes concernant les curseurs :

@@FETCH_STATUS : Renvoie l'état de la dernière instruction **FETCH** effectuée sur un curseur. Elle renvoie 0 si tout s'est bien passé, -1 s'il n'y a plus de lignes et -2 si la ligne est manquante.

@@CURSOR_ROWS : Renvoie le nombre de lignes se trouvant actuellement dans le dernier curseur ouvert. Renvoi 0 s'il n'y a pas de curseurs ouverts ou plus de ligne dans le dernier curseur. Renvoie un nombre négatif si le curseur a été ouvert de manière asynchrone.

CURSOR_STATUS : Nous permet de vérifier qu'une procédure à bien renvoyé un curseur avec un jeu de données.

Ensembliste VS Curseur

La manipulation ensembliste est juste une requête qui va nous renvoyer un ensemble de données (un resultset). C'est tout simplement des requêtes **SELECT**. Ces requêtes sont simples à effectuer bien qu'on puisse aller assez loin avec elles. Malheureusement on ne dispose pas vraiment de pouvoir sur elles, c'est la base de données qui décide ce qu'elle va nous renvoyer.

La lecture par curseur est en fait la face cachée de la manipulation ensembliste, dès que l'on fait un **SELECT**, la base de données va employer des curseurs pour construire le résultat à notre requête. Comme on vient de le voir, on peut employer ces curseurs nous-mêmes pour avoir plus de souplesses. Par contre, les curseurs sont réputés comme étant assez instables et en les manipulant nous-mêmes, on s'expose à des risques plus élevés qu'un simple **SELECT**.

Ensembliste

• Très simple à utiliser
• Aucune possibilité de modification sur le retour
• Risques quasi nuls
• Très recommandés

Curseurs

• Pas recommandé, à n'utiliser que dans des cas où l'on ne peut rien faire d'autres
• Assez complexe à utiliser
• Très puissant
• Risques d'instabilité
• Pouvoir complet sur le retour puisque c'est nous qui faisons tout

Les transactions et les verrous

Toutes les modifications de données dans SQL SERVER sont effectuées dans le cadre de transactions. Par défaut, SQL SERVER démarre une transaction pour chaque instruction individuelle et la valide automatiquement si l'exécution de l'instruction se termine normalement.

Une transaction est caractérisée les critères ACID.

– **A**tomique : Si une des instructions échoue, toute la transaction échoue.

– **C**ohérente, car la base de données est dans un état cohérent avant et après la transaction, c'est-à-dire respectant les règles de structuration énoncées.

– **I**solée : Les données sont verrouillées : il n'est pas possible depuis une autre transaction de visualiser les données en cours de modification dans une transaction.

– **D**urable : Les modifications apportées à la base de données par une transaction sont validées.

Par exemple, une transaction bancaire peut créditer un compte et en débiter un autre, ces actions devant être validées ensemble.

Les transactions utilisateurs sont implémentées sous TRANSACT SQL grâce aux instructions **BEGIN TRANSACTION**, **COMMIT TRANSACTION** et **ROLLBACK TRANSACTION.**

Les verrous empêchent les conflits de mise à jour :

– Ils permettent la sérialisation des transactions de façon à ce qu'une seule personne à la fois puisse modifier un élément de données. Dans un système de réservation de places, les verrous garantissent que chaque place n'est attribuée qu'à une seule personne.

– SQL Server définit et ajuste de manière dynamique le niveau de verrouillage approprié pendant une transaction : Il est également possible de contrôler manuellement le mode d'utilisation de certains verrous.

– Les verrous sont nécessaires aux transactions concurrentes pour permettre aux utilisateurs d'accéder et de mettre à jour les données en même temps.

Le contrôle de la concurrence permet de s'assurer que les modifications apportées par un utilisateur ne vont pas à l'encontre de celles apportées par un autre.

– Le contrôle pessimiste verrouille les données qui sont lues en vue d'une mise à jour, empêchant tout autre utilisateur de modifier ces données

– Le contrôle optimiste ne verrouille pas les données : Si les données ont été modifiées depuis la lecture, un message d'annulation est envoyé à l'utilisateur.

Le choix entre ces deux types de contrôle est effectué en fonction de l'importance du risque de conflit de données et de l'évaluation du coût de verrouillage des données (nombre d'utilisateurs, nombre de transactions, temps de réponse…). Le type de contrôle sera explicité en spécifiant le niveau d'isolement de la transaction.

BEGIN TRAN[SACTION] [nom_transaction] [WITH MARK]

Cette instruction marque le début de la transaction. Le nom de transaction n'est pas obligatoire, mais il facilite la lecture du code. L'option **WITH MARK** permet de placer le nom de la transaction dans le journal des transactions. Lors de la restauration d'une base de données à un état antérieur, la transaction marquée peut être utilisée à la place d'une date et d'une heure.

COMMIT TRAN[SACTION] [nom_transaction]

Cette instruction valide la transaction : les modifications sont effectives dans la base de données.

ROLLBACK TRAN[SACTION] [nom_transaction]

Cette instruction annule la transaction : les données de la base sont celles d'avant les ordres de modification, insertion ou suppression.

SAVE TRAN[SACTION] [nom_point_de_sauvegarde]

Cette instruction permet de définir un point d'enregistrement à l'intérieur d'une transaction.

Le code T-SQL

La première chose à faire est d'encapsuler vos requêtes dans une transaction. On va donc faire démarrer une transaction au début et la committer à la fin :

```
-- On démarre une transaction et on lui donne un nom
BEGIN TRAN changement_etat_civil
-- Vos requêtes
COMMIT TRAN changement_etat_civil
-- on commit cette transaction, c'est-à-dire qu'on valide ses modifications
```

Mais cela ne change rien au problème, en cas d'erreur, l'ensemble est tout de même committé et nous avons toujours le risque d'avoir des données incohérentes.

<u>Exemple</u> : À partir de notre base de données « Papyrus », on modifie le nom du fournisseur 120.

```
BEGIN TRAN
PRINT 'Valeur de Trancount: ' + CAST(@@TRANCOUNT AS varchar(5))
PRINT 'Avant Maj : '
SELECT NOMFOU FROM vente.FOURNISSEUR WHERE NUMFOU = 120
UPDATE vente.FOURNISSEUR SET NOMFOU = 'LEBRIGAND' WHERE NUMFOU = 120
PRINT 'Après Maj : '
```

```sql
SELECT NOMFOU FROM vente.FOURNISSEUR WHERE NUMFOU = 120
PRINT 'Valeur de Trancount : ' + CAST(@@TRANCOUNT AS varchar(5))
COMMIT TRANSACTION
```

COMMIT TRANSACTION (ou **COMMIT TRAN**) permet de valider notre transaction. **@@TRANCOUNT** retourne le nombre de transactions actives de la connexion actuelle.

Résultats | Messages

	NOMFOU
1	GROBRIGAN

	NOMFOU
1	LEBRIGAND

Résultats | Messages

```
Valeur de Trancount: 3
Avant Maj :

(1 ligne(s) affectée(s))

(1 ligne(s) affectée(s))
Après Maj :

(1 ligne(s) affectée(s))
Valeur de Trancount : 2
```

```sql
COMMIT TRAN -- Pour valider ma transaction
-- OU
ROLLBACK TRAN -- Pour annuler ma transaction
```

Si l'une ou l'autre des deux instructions est absente de la transaction, vous ne pourrez plus interroger la table en question tant que la transaction n'est pas terminée.

Verrouillages dans SQL Server

Lors de transactions concurrentes, des verrous peuvent être posés pour éviter les situations suivantes :

– **Mise à jour perdue** : Une mise à jour peut être perdue lorsqu'une transaction écrase les modifications effectuées par une autre transaction.

– **Lecture incorrecte :** Se produit lorsqu'une transaction lit des données non validées provenant d'une autre transaction.

– **Lecture non renouvelable :** Se produit lorsque, une transaction devant lire la même ligne plusieurs fois, la ligne est modifiée par une autre transaction et donc produit des valeurs différentes à chaque lecture.

– **Lectures fantômes :** Se produit lorsqu'une autre transaction insert une ligne au sein de la transaction en cours.

Un **verrou partagé** appliqué à une ressource par une première transaction autorise l'acquisition par une deuxième transaction d'un verrou partagé sur cette ressource, même si la première transaction n'est pas terminée. Un verrou partagé sur une ligne est libéré dès la lecture de la ligne suivante.

Un verrou partagé est alloué pour une instruction **SELECT**.

SQL Server utilise des **verrous exclusifs** lorsqu'une instruction **INSERT**, **DELETE** ou **UPDATE** est exécutée.

Une seule transaction peut acquérir un verrou exclusif sur une ressource ; une transaction ne peut pas acquérir un verrou exclusif sur une ressource tant qu'il existe des verrous partagés ; Une transaction ne peut pas acquérir un verrou partagé sur une ressource déjà pourvue d'un verrou exclusif.

SQL Server permet de définir un niveau d'isolement qui protège une transaction vis-à-vis des autres.

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
REPEATABLE READ | SNAPSHOT | SERIALIZABLE}
```

Read UnCommitted : Indique à SQL Server de ne pas placer de verrous partagés : Une transaction peut lire des données modifiées non encore validées par une autre transaction.
Des lectures incorrectes peuvent se produire.

Read Committed (Option par défaut) : Indique à SQL Server d'utiliser des verrous partagés pendant la lecture : une transaction ne peut pas lire les données modifiées, mais non validées d'une autre transaction. La lecture incorrecte ne peut pas se produire.

Repeatable Read : SQL Server place des verrous partagés sur toutes les données lues par chaque instruction de la transaction et les maintient jusqu'à la fin de la transaction : Une transaction ne peut pas lire des données modifiées, mais pas encore validées par une autre transaction, et ne peut modifier les données lues par la transaction active tant que celle-ci n'est pas terminée.

La lecture incorrecte et la lecture non renouvelable ne peuvent pas se produire.

Snapshot : Une transaction n'a pas accès aux modifications de données apportées par une autre transaction : Les données vues par la première transaction sont les données antérieures au début de la deuxième transaction.

Serializable : Une transaction ne peut pas lire des données modifiées, mais pas encore validées par une autre transaction, et ne peut modifier les données lues par la transaction active tant que celle-ci n'est pas terminée.

Une transaction ne peut pas insérer de nouvelles lignes avec des valeurs de clés comprises dans le groupe de clés lues par des instructions de la transaction active, tant que celle-ci n'est pas terminée.

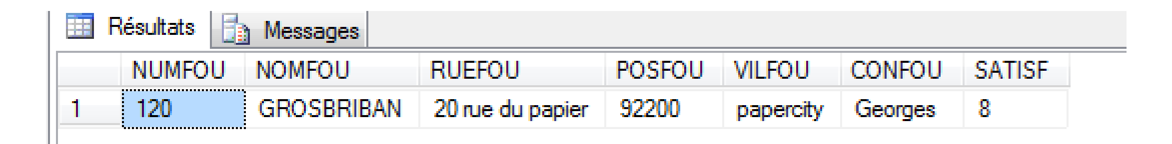
La lecture incorrecte la lecture non renouvelable et les lectures fantômes ne peuvent pas se produire.

Exemple : La transaction de l'exemple suivant n'est pas terminée, mais nous avons utilisé l'option **READ UNCOMMITTED**.

```
USE Papyrus
GO
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
GO
BEGIN TRAN
PRINT 'Valeur de Trancount: ' + CAST(@@TRANCOUNT AS varchar(5))
PRINT 'Avant Maj : '
SELECT NOMFOU FROM vente.FOURNISSEUR WHERE NUMFOU = 120
UPDATE vente.FOURNISSEUR SET NOMFOU = 'GROSBRIBAN' WHERE NUMFOU = 120
PRINT 'Après Maj : '
SELECT NOMFOU FROM vente.FOURNISSEUR WHERE NUMFOU = 120
PRINT 'Valeur de Trancount : ' + CAST(@@TRANCOUNT AS varchar(5))
```

Nous interrogeons notre table FOURNISSEUR.

```
SELECT *
FROM vente.FOURNISSEUR
WHERE NUMFOU = 120
```



	NUMFOU	NOMFOU	RUEFOU	POSFOU	VILFOU	CONFOU	SATISF
1	120	GROSBRIBAN	20 rue du papier	92200	papercity	Georges	8

Cependant, on peut toujours annuler la transaction ou la valider par les instructions **COMMIT TRAN** ou **ROLLBACK TRAN**.

Le niveau d'isolement spécifie le comportement de verrouillage par défaut de toutes les instructions de la session (connexion).

Plus le niveau d'isolement est élevé, plus les verrous sont maintenus longtemps et plus ils sont restrictifs.

La commande **DBCC USEROPTIONS** renvoie entre autres, le niveau d'isolement de la session.
Il est possible de définir la durée maximale pendant laquelle SQL Server permet à une transaction d'attendre le déverrouillage d'une ressource bloquée. La variable globale **@@lock_timeout** donne le temps d'attente défini.

SET LOCK_TIMEOUT *délai_attente* positionne le délai, en millisecondes, avant que ne soit renvoyé un message d'erreur (-1 indique qu'il n'y a pas de délai par défaut).

Gestion des erreurs

La gestion des erreurs se fait dans un bloc **BEGIN TRY** / **END TRY**. En cas d'erreurs, c'est le bloc **BEGIN CATCH** / **END CATCH** qui gère l'erreur. Sur notre base de données « Papyrus », on crée une transaction qui permet d'insérer des lignes sur les tables **ENTCOM** et **LIGCOM**. Si la transaction se passe bien, elle est validée, sinon elle est annulée.

```
-- On démarre une transaction et on lui donne un nom
BEGIN TRANSACTION transaction_CdeBande
-- Positionne le délai, en millisecondes
SET LOCK_TIMEOUT 2000
-- On déclare également la variable suivante
DECLARE @pcde int
BEGIN TRY
-- Opération d'insertion dans lignes de ventes
INSERT INTO [vente].[ENTCOM](OBSCOM,NUMFOU)
      VALUES('Bande 09180',09180)
SELECT @pcde =@@IDENTITY
-- Commande de bande magnétique au fournisseur 09180
INSERT INTO [vente].[LIGCOM](NUMCOM,NUMLIG, CODART,QTECDE, PRIUNI, DERLIV)
     VALUES(@pcde, 01, 'B001', 200, 140, Dateadd(mm,-1,getdate()) )
INSERT INTO [vente].[LIGCOM](NUMCOM,NUMLIG, CODART,QTECDE, PRIUNI, DERLIV)
     VALUES(@pcde, 02, 'B002', 200, 140, Dateadd(mm,-2,getdate()) )
PRINT 'Transaction réussie'
COMMIT TRAN
END TRY
BEGIN CATCH
PRINT 'Transaction annulée'
ROLLBACK TRAN
END CATCH
```

Points d'enregistrements

Une autre chose qu'il est utile de connaître est le fait qu'on peut insérer des savepoint dans une transaction, c'est-à-dire un point d'enregistrement, sur lequel on peut se baser pour faire un **ROLLBACK**. Ainsi, on n'a pas besoin de faire une annulation sur l'intégralité de la transaction, si on a deux parties bien distinctes dans notre script, on peut intercaler un point d'enregistrement entre les 2. Si la première partie s'est déroulée sans problèmes, mais que la deuxième partie s'est mal passée, on faire un **ROLLBACK** jusqu'à notre point d'enregistrement puis un commit. Ainsi, tout ce qui s'est bien passé est validé et le reste est annulé.

Pour sauvegarder un point d'enregistrement, il vous suffit de faire ceci :

```
-- On sauvegarde un point d'enregistrement pour la transaction
SAVE TRANSACTION nom_du_savepoint
```

Et si ensuite, vous voulez revenir à cet état, il vous suffit de faire :

```
-- On annule toutes les modifications de cette transaction jusqu'au point
d'enregistrement
ROLLBACK TRANSACTION nom_du_savepoint
```

Les déclencheurs

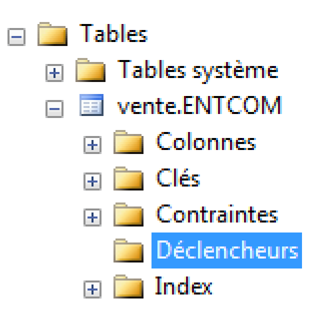
Les déclencheurs sont des objets anciens de SQL Server, qui permettent d'effectuer automatiquement des tâches administratives de maintien de la base de données. En effet, les déclencheurs, ou triggers, vont nous permettre, dans leur généralité, d'effectuer des opérations automatiques, liées à une opération particulière sur un objet de la base, par l'utilisateur. Plus précisément, lorsqu'un utilisateur va appliquer une opération du DML par exemple, sur une table, si un déclencheur est défini sur cette table, une action automatique en rapport avec l'action de l'utilisateur s'exécutera. On commence alors de parler de la notion de programmation événementielle, en rapport au fait qu'un événement en déclenche un autre.

Les déclencheurs du DML

Les déclencheurs sont donc des objets de la base qui nous permettent d'automatiser des actions. Quelle est la particularité des déclencheurs du DML ? Ils vont nous permettre d'effectuer une instruction, à chaque fois qu'une instruction du DML est effectuée pour une table donnée. Prenons un exemple pour mieux comprendre : nous avons deux tables qui fonctionnent ensemble, table1 et table2. Nous définissons un déclencheur du DML sur table1, qui fait en sorte que, si la table table1 est mise à jour, les valeurs correspondantes dans table2 sont mises à jour de la même manière. C'est de cette manière que nous allons automatiser les actions sur les objets de la base. Pour une instruction donnée sur un objet, une action liée s'exécutera à la suite.

Création d'un déclencheur du DML

La création de ce genre d'objet de la base ne peut se faire que par du code T-SQL, en revanche, il est possible, comme pour tout autre objet de la base d'auto générer le modèle de création d'un déclencheur. Pour ce faire, étendez les nœuds de l'explorateur d'objet jusqu'à la table pour laquelle vous voulez créer le déclencheur, puis étendez la table choisie comme suit :



Une fois cette opération effectuée, faites un clic droit sur le nœud « **Déclencheurs** » et choisissez l'option « **Nouveau déclencheur …** ». Un code auto généré apparaît alors, dans une nouvelle fenêtre de requêtes. Voici ce code :

```
CREATE TRIGGER <Schema_Name, sysname, Schema_Name>.<Trigger_Name, sysname,
Trigger_Name>
ON <Schema_Name, sysname, Schema_Name>.<Table_Name, sysname, Table_Name>
(AFTER | FOR | INSTEAD OF) <Data_Modification_Statements, ,
INSERT,DELETE,UPDATE>
[WITH APPEND][NOT FOR REPLICATION]
AS
BEGIN
-- SET NOCOUNT ON added to prevent extra result sets from
-- interfering with SELECT statements.
SET NOCOUNT ON;
-- Insert statements for trigger here
```

Expliquons ce code. Tout d'abord, l'instruction **CREATE TRIGGER** est l'instruction DDL qui va nous permettre de créer un déclencheur. Il est nécessaire de préciser le nom du déclencheur à la suite de cette instruction, puisque comme tout objet de la base, un déclencheur peut être mentionné dans du code seulement grâce à son nom unique. La clause **ON** va nous permettre de définir sur quelle table nous allons définir le déclencheur que nous créons et les clauses **AFTER**, **INSTEAD OF** nous permettront par la suite, de données les conditions d'action du déclencheur.

- AFTER : Le déclencheur s'exécutera après insertion, mise à jour ou suppression dans la table qui contient le déclencheur.

- INSTEAD OF : Le déclencheur s'exécutera avant insertion, mise à jour ou suppression dans la table qui contient le déclencheur.

Des options sont disponibles pour les triggers du DML, et elles sont les suivantes :

- WITH APPEND : cette option permet d'ajouter plusieurs déclencheurs sur un même objet et un même ordre SQL. Ceci est dû à un comportement différent des déclencheurs au passage de la version compatible 65 et 70.

- NOT FOR REPLICATION : Le déclencheur défini avec cette option ne sera pas pris en compte dans un processus de modification des données par réplication.

Par exemple, après insertion dans la table précisée, faire ceci. Comme pour une fonction, les délimiteurs des actions dans un déclencheur sont les clauses **AS BEGIN** et **END**. Tout le code présent entre ces deux délimiteurs sera exécuté, à la seule condition que l'action déclenchant le déclencheur soit faite auparavant. Prenons maintenant un exemple concret pour illustrer nos propos. Dans le code du script proposé en annexe de ce cour, il y a un déclencheur du DML nous allons l'utiliser ici comme exemple.

Remarque : Les instructions suivantes ne sont pas autorisées dans le corps du déclencheur, à savoir **CREATE**, **DROP**, **ALTER**, **TRUNCATE**, **GRANT**, **REVOKE**, **UPDATE STATISTICS**, **RECONFIGURE** et **LOAD**.

```
CREATE TRIGGER Soustraction_Stock
ON Commande
AFTER INSERT
AS
BEGIN
DECLARE @Id_Stock int, @Quantite int
SELECT @Id_Stock = Id_stock FROM INSERTED
SELECT @Quantite = Quantite FROM INSERTED
UPDATE [Stock]
SET [Quantite] = [Quantite]-@Quantite
WHERE Id_Stock = @Id_Stock
END
```

Au travers de ce code T-SQL, nous créons un déclencheur, dont le nom est Soustraction_Stock, sur la table commande. Nous y associons l'événement **AFTER INSERT**, qui précise qu'après insertions de données dans la table Commande, le code contenu entre les délimiteurs **AS BEGIN** et **END** doit être exécuté. Le code contenu entre les délimiteurs, dans cet exemple, récupère certaines valeurs insérées dans la table Commande, en l'occurrence Id_Stock et Quantité, dans des variables déclarées en début de lot grâce aux mots clé **FROM INSERTED**, et soustrait la quantité de la commande, à la quantité du stock, en fonction d'un Id_Stock donné, grâce à un **UPDATE** de la table Stock. Les trois mots clé **FROM UPDATED**, **FROM INSERTED**, **FROM DELETED** nous permettent de récupérer les valeurs insérées, mises à jour ou supprimées, avant l'activation du déclencheur.

Les instructions de déclencheur DML utilisent deux tables spéciales : la table **deleted** et la table **inserted**. SQL Server les crée et les gère automatiquement. Ces tables temporaires résidant en mémoire servent à tester les effets de certaines modifications de données et à définir des conditions pour les actions de déclenchement DML.

Exemple 1 : Création d'un déclencheur **AFTER DELETE**. À partir de notre base de données « Papyrus », créer un déclencheur **AFTER DELETE** sur la table VENTE, pour empêcher l'utilisateur de supprimer plusieurs lignes de ventes à la fois.

Nous allons étudier deux possibilités. La première est très simple. Nous utilisons la fonction **@@ROWCOUNT**.

```
-- S'il existe déjà un trigger de ce nom
IF OBJECT_ID ('vente.Supp_Lignes_Vendre', 'TR') IS NOT NULL
   -- Alors on le supprime
   DROP TRIGGER vente.Supp_Lignes_Vendre;
GO
CREATE TRIGGER vente.Supp_Lignes_Vendre
-- Agit sur la table Vendre
ON vente.VENDRE
-- On précise qu'il s'agit d'un déclencheur pour DELETE
FOR DELETE
AS
BEGIN
      IF @@ROWCOUNT > 1
            BEGIN
            -- On indique l'erreur
            RAISERROR ('Trop de lignes à supprimer !', 10, 1)
            -- Si erreur, on annule la transaction
            ROLLBACK TRAN
```

```
            -- On arrête le script
            RETURN
      END
END
```

@@ROWCOUNT Retourne le nombre de lignes affectées par la dernière instruction. Si le nombre de lignes est supérieur à 2 milliards, utilisez **@@ROWCOUNT_BIG**.

Si nous testons ce trigger, on ne peut pas supprimer plusieurs lignes.

```
USE Papyrus
DELETE FROM vente.VENDRE
WHERE CODART like 'T%'
```

En revanche, on peut supprimer une ligne.

```
USE Papyrus
DELETE FROM vente.VENDRE
WHERE CODART = 'T001' AND NUMFOU = '9180'
```

La deuxième possibilité n'est pas à privilégier, mais nous permet d'introduire la notion de curseur :

```
CREATE TRIGGER vente.Supp_Lignes_Vendre
-- Agit sur la table Vendre
ON vente.VENDRE
-- On précise qu'il s'agit d'un déclencheur pour DELETE
FOR DELETE
AS
-- On déclare nos variables
DECLARE @codart char(4)
DECLARE @nbrligne int
-- On désigne curseur_suppression le nom du curseur ouvert grâce auquel
s'effectue l'extraction
DECLARE curseur_suppression CURSOR FOR
-- Le cursor agit pour la sélection suivante
SELECT CODART, COUNT(*) FROM DELETED GROUP BY CODART
/* Ouvre un curseur de serveur Transact-SQL et remplit ce dernier en
exécutant
l'instruction Transact-SQL spécifiée dans l'instruction DECLARE CURSOR ou
SET
cursor_variable. */
OPEN curseur_suppression
-- Utilisation de FETCH : Récupère une ligne spécifique d'un curseur
FETCH curseur_suppression INTO @codart, @nbrligne
/* Into : Permet aux données issues des colonnes d'une extraction d'être
placées
dans des variables locales. Chaque variable de la liste (de gauche à
droite) est
associée à la colonne correspondante dans le jeu de résultats du curseur.
Le type
de données de chaque variable doit correspondre ou être une conversion
implicite
du type de données de la colonne du jeu de résultats correspondante. Le
nombre
de variables doit correspondre au nombre de colonnes dans la liste de
sélection
du curseur. */
-- Tant que @@FETCH_STATUS = 0
WHILE @@FETCH_STATUS = 0
```

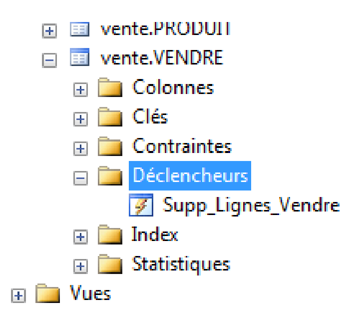
```sql
/* @@FETCH_STATUS retourne l'état de la dernière instruction FETCH
effectuée sur un
curseur actuellement ouvert par la connexion. */
BEGIN
IF @nbrligne > 1
BEGIN
-- On indique l'erreur
RAISERROR ('Trop de lignes à supprimer pour article %s', 10, 1,@codart)
-- Si erreur, on annule la transaction
ROLLBACK TRAN
RETURN
END
FETCH curseur_suppression INTO @codart, @nbrligne
END
-- On ferme le curseur
CLOSE curseur_suppression
/* Supprime une référence de curseur. Une fois la dernière référence de
curseur désallouée, MicrosoftSQL Server libère les structures de données
contenant le curseur. */
DEALLOCATE curseur_suppression
```

Testons ce trigger :

```sql
USE Papyrus
DELETE FROM vente.VENDRE
WHERE CODART like 'I%'
```

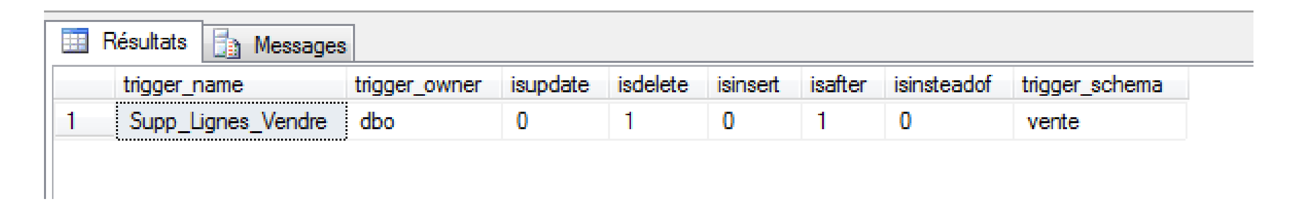
```
Messages
Trop de lignes à supprimer pour article I100
Msg 3609, Niveau 16, État 1, Ligne 2
La transaction s'est terminée dans le déclencheur. Le traitement a été abandonné.
```

Il est possible de voir les déclencheurs créés, en déployant les nœuds des tables prévus à cet effet, comme montrés ci-dessous :



Ou encore en utilisant la procédure stockée **sp_helptrigger**. L'exemple suivant exécute sp_helptrigger pour produire des informations sur le ou les déclencheurs de la table vente.VRENDRE.

```sql
USE Papyrus;
GO
EXEC sp_helptrigger 'vente.VENDRE';
```



	trigger_name	trigger_owner	isupdate	isdelete	isinsert	isafter	isinsteadof	trigger_schema
1	Supp_Lignes_Vendre	dbo	0	1	0	1	0	vente

Exemple 2 : Création d'un déclencheur **AFTER UPDATE**. Pour le besoin de notre exemple, nous créons une table ARTICLES_A_COMMANDER. Lorsque le stock physique augmenté de la somme des quantités de l'article dans la table ARTICLES_A_COMMANDER et de la somme des quantités de l'article dans la table LIGCDE devient inférieur au stock d'alertes, une nouvelle ligne est insérée dans la table ARTICLES_A_COMMANDER.

Lorsque le stock physique est inférieur à 0, toute décrémentation est rejetée : la transaction est annulée et un message est envoyé dans les journaux.

Nous créons la table ARTICLES_A_COMMANDER de structure : ARTICLES_A_COMMANDER (CODART, QTE, DATECOM).

```
CREATE TABLE [vente].[ARTICLES_A_COMMANDER](
[CODART] [char](4)NOT NULL ,
[QTE] [smallint] NOT NULL,
[DATECOM] [date] NOT NULL)
```

Créer un déclencheur **UPDATE** sur la table PRODUIT de notre base de données « Papyrus » :

```
-- S'il existe déjà un trigger de ce nom
IF OBJECT_ID ('vente.ARTICLES_TRIG', 'TR') IS NOT NULL
   -- Alors on le supprime
   DROP TRIGGER vente.ARTICLES_TRIG;
GO
/* Lorsque le stock physique augmenté de la somme des quantités de
l'article dans la
table ARTICLES_A_COMMANDER et de la somme des quantités de l'article dans
la table
LIGCDE devient inférieur au stock d'alerte, une nouvelle ligne est insérée
dans la
table ARTICLES_A_COMMANDER ;
> CODART = CODART
> DATE = date du jour (par défaut)
> QTE = à définir */
CREATE TRIGGER vente.ARTICLES_TRIG
-- Agit sur la table PRODUIT
ON vente.PRODUIT
-- On précise qu'il s'agit d'un déclencheur pour UPDATE
FOR UPDATE
AS
-- On déclare nos variables
DECLARE @stockphysique int, @stockalerte int, @totalqtecommandee int,
@totalqte int
-- On incrémente notre variable @stockphysique et notre variable
@stockalerte
SELECT @stockphysique = p.STKPHY, @stockalerte = p.STKALE
FROM PRODUIT p
INNER JOIN inserted i
ON i.CODART = p.CODART
-- On ajoute une condition, si stocke est négatif
/* Lorsque le stock physique est inférieur à 0, toute décrémentation est
rejetée :
La transaction est annulée et un message est envoyé dans les journaux. */
```

```
IF (@stockphysique <= 0)
BEGIN
-- On indique l'erreur
RAISERROR ('Le stock est négatif !',16,1)
-- Si erreur, on annule la transaction
ROLLBACK TRAN
END
-- On incrémente notre variable @totalqtecommandee
SET @totalqtecommandee = isnull((select SUM(QTECDE)
FROM vente.LIGCOM l
INNER JOIN inserted i
ON l.CODART = i.CODART
GROUP BY l.CODART),0)
-- On incrémente notre variable @totalqte
SET @totalqte = isnull((select SUM(QTE)
FROM ARTICLES_A_COMMANDER a
INNER JOIN inserted i
ON a.CODART = i.CODART
GROUP BY a.CODART),0)
IF ((@stockphysique + @totalqtecommandee + @totalqte) < @stockalerte)
-- Alors on insère dans la table ARTICLES_A_COMMANDER
INSERT INTO ARTICLES_A_COMMANDER (CODART, QTE, DATECOM)
-- La sélection suivante
SELECT p.CODART, @stockalerte-@stockphysique + @totalqtecommandee +
@totalqte, getdate()
FROM PRODUIT p
INNER JOIN Inserted i
ON p.CODART = i.CODART
```

Pour comprendre le problème : Soit l'article « I120 », de stock d'alerte = 5, les tableaux ci-dessous présentent les modifications de la table PRODUIT et les actions à effectuer sur la table ARTICLES_A_COMMANDER, en fonction de la quantité en commande dans la table LIGCOM.

Jeu d'essai n° 1 :

PRODUIT			LIGCOM	ARTICLES_A_COMMANDER	
STKPHY avant	-	STKPHY après	QTECDE		QTE
25	5	20	0	aucune action	
20	15	5		aucune action	
5	1	4		insertion	1
4	2	2		Insertion	2
2	2	0		Insertion	2
0	2	-2		rejet	

Jeu d'essai n° 2 :

PRODUIT			LIGCOM	ARTICLES_A_COMMANDER	
STKPHY avant	-	STKPHY après	QTECDE		QTE
25	5	20	3	aucune action	
20	15	5		aucune action	
5	1	4		aucune action	
4	3	1		Insertion	1
1	1	0		Insertion	1
0	2	-2		rejet	

Nous testerons les différents je d'essaie avec la requête suivante :

```
USE Papyrus
UPDATE vente.PRODUIT
SET STKPHY = STKPHY - 5
WHERE CODART = 'I120'
```

Exemple 3 : L'exemple suivant exécute **sp_helptrigger** pour produire des informations sur le ou les déclencheurs de la table vente.VENDRE.

```
USE Papyrus;
GO
EXEC sp_helptrigger 'vente.VENDRE';
```

	trigger_name	trigger_owner	isupdate	isdelete	isinsert	isafter	isinsteadof	trigger_schema
1	Supp_Lignes_Vendre	03112-375\Util5	0	1	0	1	0	vente

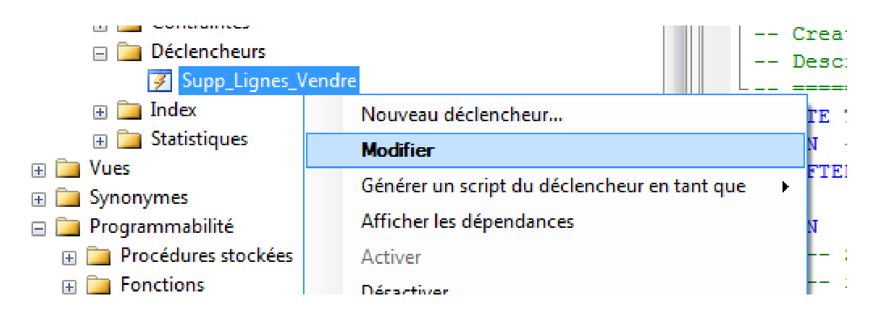
Modification d'un déclencheur du DML

Comme pour tout objet de la base, le déclencheur peut être modifié de deux manières, par code T-SQL ou encore par l'interface graphique. Voyons les deux manières.

La structure générale de modification par code T-SQL est la même que pour n'importe quel objet de la base puisque nous allons utiliser **ALTER.** Voyons un exemple :

```
USE [Papyrus]
GO
/****** Object:  Trigger [vente].[Supp_Lignes_Vendre]    Script Date:
01/27/2011 22:11:02 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER TRIGGER [vente].[Supp_Lignes_Vendre]
ON [vente].[VENDRE]
FOR DELETE
AS
DECLARE @codart char(4)
DECLARE @nbrligne int
DECLARE curs_del CURSOR FOR
SELECT CODART, COUNT(*) FROM DELETED GROUP BY CODART
OPEN curs_del
FETCH curs_del INTO @codart, @nbrligne
WHILE @@FETCH_STATUS = 0
BEGIN
IF @nbrligne > 1
BEGIN
RAISERROR ('Trop de lignes à supprimer pour article %s', 10, 1,@codart)
ROLLBACK TRAN
RETURN
END
FETCH curs_del INTO @codart, @nbrligne
END
CLOSE curs_del
DEALLOCATE curs_del
```

Par l'interface graphique, il vous faudra tout de même utiliser le code Transact-SQL. Étendez les nœuds qui mènent à la table qui contient vos déclencheurs, comme ceci :



Effectuez alors un clic droit sur le déclencheur et sélectionnez l'option « **Modifier** ». Il ne vous reste qu'à modifier le code.

SQLQuery3.sql - (...DELTA1\O.S.G (55)) | SQLQuery2.sql - (...DELTA1\O.S.G (53)) | SQLQuery1.sql - (l...ELTA1\O.S.G

```
USE [Papyrus]
GO
/****** Object:  Trigger [vente].[Supp_Lignes_Vendre]    Script Date: 01/27
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER TRIGGER [vente].[Supp_Lignes_Vendre]
ON [vente].[VENDRE]
FOR DELETE
AS
DECLARE @codart char(4)
DECLARE @nbrligne int
DECLARE curs_del CURSOR FOR
SELECT CODART, COUNT(*) FROM DELETED GROUP BY CODART
OPEN curs_del
FETCH curs_del INTO @codart, @nbrligne
WHILE @@FETCH_STATUS = 0
BEGIN
IF @nbrligne > 1
BEGIN
RAISERROR ('Trop de lignes à supprimer pour article %s', 10, 1,@codart)
ROLLBACK TRAN
RETURN
END
FETCH curs_del INTO @codart, @nbrligne
END
CLOSE curs_del
DEALLOCATE curs_del
```

Suppression d'un déclencheur du DML

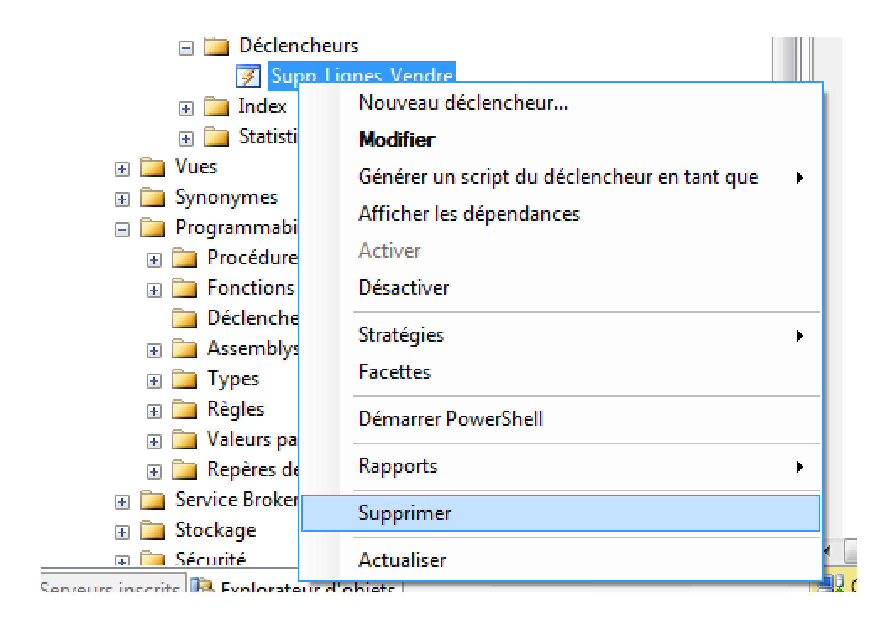
Comme pour tout objet de la base, le déclencheur peut être supprimé de deux manières, par code T-SQL ou encore par l'interface graphique. Voyons les deux manières.

La structure générale de suppression par code T-SQL est la même que pour n'importe quel objet de la base puisque nous allons utiliser **DROP.** Voyons un exemple :

```
DROP TRIGGER Supp_Lignes_Vendre
```

Il suffit simplement d'ajouter le mot clé **TRIGGER** pour spécifier que nous allons supprimer un déclencheur, puis le nom de l'objet à supprimer.

Par l'interface graphique, étendez les nœuds qui mènent à la table qui contient vos déclencheurs, comme ceci :



Effectuez alors un clic droit sur le déclencheur et sélectionnez l'option « **Supprimer** ».

Les déclencheurs du DDL

SQL Server propose des déclencheurs du DDL. Ce type de déclencheur fonctionne de la même manière que les déclencheurs du DML, simplement, les objets affectés par ces déclencheurs et les instructions auxquelles ils réagissent ne sont pas les mêmes. En effet, un trigger du DDL agit au niveau d'une base de données ou du serveur où il est défini, alors qu'un déclencheur du DML agit simplement pour la table ou la vue sur laquelle il est défini. Dans ce dernier cas, le déclencheur est stocké dans la base de données MASTER. De plus, ceux du type DML réagissent à des instructions du DML, alors ceux du DDL réagissent à des instructions du DDL, telles que **CREATE**, **ALTER**, **DROP**, **GRANT**, **REVOKE**, **DENY** et **UPDATE STATISTICS**. Comme pour ceux du DML, les déclencheurs du DDL sont exécutés après l'instruction DDL qui leur est attachée. Le but principal de ce type d'objet est de gérer l'ajout et la suppression ou encore la mise à jour d'objets dans la table, ou bien d'automatiser celle-ci. Enfin, et c'est la dernière différence entre les deux différents types de déclencheurs que sont les DML et les DDL, il n'est pas possible, contrairement à un DML, de créer un déclencheur **INSTEAD OF** de type DDL.

Création d'un déclencheur du DDL

Présentons maintenant la structure générale de création d'un déclencheur de type DDL :

```
USE Entreprise
GO
CREATE TRIGGER nom_trigger
ON DATABASE
FOR CREATE_TABLE
AS
BEGIN
ROLLBACK TRAN
END
```

L'instruction **CREATE TRIGGER** nous permet d'annoncer que nous allons créer un déclencheur. Il est nécessaire de préciser son nom à la suite de cette instruction. Par la suite,

on définit si ce déclencheur est défini pour une base de données en particulier ou pour tout le serveur grâce aux mots clé **ALL SERVER** et **DATABASE**. Nous définirons ensuite les options **WITH ENCRYPTION** et **EXECUTE AS**. La clause **AFTER** permet, comme pour un déclencheur du DML, de préciser après quelle instruction le déclencheur doit être activé. Ces instructions seront présentées ci-après, dans un tableau résumé. Les clauses **AS BEGIN** et **END** définissent le corps du déclencheur, c'est-à-dire les actions à effectuer dans le cas où celui-ci est activé.

Voici les instructions disponibles pour un déclencheur du DDL (ces instructions ont deux cibles distinctes, les objets du serveur ou les objets de la base. Nous détaillerons les deux) :

Les instructions sur une base de données :

- **Tables :**

DDL_TABLE_EVENTS
CREATE TABLE
ALTER_TABLE
DROP_TABLE

- **Vues :**

CREATE_VIEW
ALTER_VIEW
DROP_VIEW

- **Synonymes :**

CREATE_SYNONYM
DROP_SYNONYM

- **Fonctions :**

CREATE_FUNCTION
ALTER_FUNCTION
DROP_FUNCTION

- **Procédures :**

CREATE_PROCEDURE
ALTER_PROCEDURE
DROP_PROCEDURE

- **Déclencheurs :**

CREATE_TRIGGER
ALTER_TRIGGER
DROP_TRIGGER

- **Notifications :**

CREATE_EVENT_NOTIFICATION
DROP_EVENT_NOTIFICATION

- **Index :**

CREATE_INDEX
ALTER_INDEX
DROP_INDEX

- **Statistiques :**

CREATE_STATISTICS
UPDATE_STATISTICS
DROP STATISTICS

- **Assemblies :**

CREATE_ASSEMBLY
ALTER_ASSEMBLY
DROP_ASSEMBLY

- **Types :**

CREATE_TYPE
DROP_TYPE

- **Users :**

CREATE_USER
ALTER_USER
DROP_USER
CREATE_ROLE
ALTER_ROLE
DROP_ROLE

- **Rôles :**

CREATE_APPLICATION_ROLE
ALTER_APPLICATION_ROLE
DROP_APPLICATION_ROLE

- **Schémas :**

CREATE_SCHEMA

ALTER_SCHEMA
DROP_SCHEMA

- **Messages Types :**

CREATE_MESSAGE_TYPE
ALTER_MESSAGE_TYPE
DROP_MESSAGE_TYPE

- **Contrats :**

CREATE_CONTRACT
ALTER_CONTRACT
DROP_CONTRACT

- **Queues :**

CREATE_QUEUE
ALTER_QUEUE
DROP_QUEUE

- **Services :**

CREATE_SERVICE

ALTER_SERVICE
DROP_SERVICE

- **Route :**

CREATE_ROUTE
ALTER_ROUTE
DROP_ROUTE

- **Service Binding :**

CREATE_REMOTE_SERVICE_BINDING
ALTER_REMOTE_SERVICE_BINDING
DROP_REMOTE_SERVICE_BINDING

- **Droits sur la base de données :**

GRANT_DATABASE
DENY_DATABASE
REVOKE_DATABASE

- **Secexprer :**

CREATE_SECEXPR
DROP_SECEXPR

- **XML :**

CREATE_XML_SCHEMA
ALTER_XML_SCHEMA
DROP_XML_SCHEMA

- **Fonction de partition :**

CREATE_PARTITION_FUNCTION
ALTER_PARTITION_FUNCTION
DROP_PARTITION_FUNCTION

- **Schémas de partition :**

CREATE_PARTITION_SCHEME
ALTER_PARTITION_SCHEME
DROP_PARTITION_SCHEME

Les instructions sur le serveur :

- **Logins :**

CREATE_LOGIN
ALTER_LOGIN
DROP_LOGIN

- **Endpoints :**

CREATE_HTTP_ENDPOINT
DROP_HTTP_ENDPOINT

- **Droits d'accès au serveur :**

GRANT_SERVER_ACCESS

DENY_SERVER_ACCESS
REVOKE_SERVER_ACCESS

- **Certificats :**

CREATE_CERT
ALTER_CERT
DROP_CERT

La liste ci-dessus n'est pas exhaustive. Voyons maintenant ce que signifient les options disponibles :

- WITH ENCRYPTION : Permet de crypter le code du déclencheur dans la base de données MASTER.

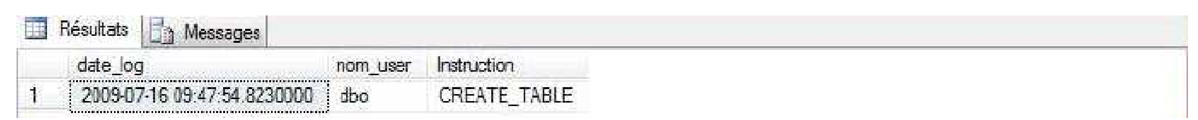
Prenons un exemple concret de création d'un déclencheur du DDL.

```
USE Entreprise
GO
CREATE TABLE log_database
(date_log datetime2, nom_user nvarchar(50), Instruction nvarchar(100))
GO
CREATE TRIGGER trigger_log
ON DATABASE
AFTER DDL_TABLE_EVENTS
AS
BEGIN
INSERT INTO log_database
(date_log, nom_user, Instruction)
VALUES
(GETDATE(),
CURRENT_USER,
EVENTDATA().value('(/EVENT_INSTANCE/EventType) [1]','nvarchar(50)'))
END
```

Dans ce cas-là, pour chaque instruction du DDL faite pas un utilisateur quelconque, nous allons insérer dans la table log_database la date, l'utilisateur et le type d'évènements qu'il a utilisé, pour constituer au final, une sorte de registre des actions DDL faites sur les tables. La dernière ligne de l'instruction **INSERT** (EVENTDATA().value('(/EVENT_INSTANCE/EventType) [1]','nvarchar(50)')) est un travail sur un fichier XML. Retenez simplement qu'il s'agit d'une méthode d'extraction d'information d'un fichier XML.

Voilà ce qu'il se passe lorsque nous créons une table, et que nous exécutons le code suivant :

```
SELECT * FROM dbo.log_database
```



Résultats | Messages

	date_log	nom_user	Instruction
1	2009-07-16 09:47:54.8230000	dbo	CREATE_TABLE

Un autre type de déclencheur existe, même si on ne peut pas parler directement de type de déclencheur puisqu'il fait partie des déclencheurs du DDL. C'est le type des triggers de connexion. Ils se présentent de la même manière qu'un déclencheur DDL simple, seule une différence est à noter. La présence du mot clé **FOR LOGON**, comme dans la structure suivante :

```
CREATE TRIGGER trigger_log
```

```
ON ALL SERVER
FOR LOGON
AS
BEGIN
-- Instructions
END
```

Il faut faire très attention avec les déclencheurs de connexions, car dans certains cas, ils peuvent vous empêcher une connexion au serveur.

Exemple 1 : Création d'un déclencheur DDL. Créer un déclencheur qui interdit toutes opérations de création, modification, suppression sur les triggers.

```
CREATE TRIGGER securite_trigger0
ON DATABASE
FOR CREATE_TRIGGER, ALTER_TRIGGER, DROP_TRIGGER
AS
RAISERROR('Création / Modification / Suppression impossible !',16,1)
ROLLBACK TRAN
```

Exemple 2 : Création d'un trigger qui se déclenche à la modification / suppression de lignes d'une table.

```
CREATE TRIGGER securite_trigger1
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
RAISERROR('Modification / Suppression impossible',16,1)
ROLLBACK TRAN
```

Un déclencheur DDL peut également être programmé pour s'exécuter sur un groupe prédéfini d'instructions Transact-SQL.

Exemple 3 : Création d'un trigger qui se déclenche lors d'un évènement quelconque affectant les tables.

```
CREATE TRIGGER securite_trigger2
ON DATABASE
FOR DDL_TABLE_VIEW_EVENTS
AS
RAISERROR('Modification / Suppression impossible',16,1)
ROLLBACK TRAN
```

On peut utiliser la fonction **EVENTDATA** pour extraire le texte de l'instruction Transact-SQL correspondante.

Exemple 4 : Le déclencheur DML suivant affiche un message à destination de l'utilisateur lorsque quelqu'un essaye d'ajouter ou de modifier des données dans la table PRODUIT.

```
USE Papyrus;
GO
-- S'il existe déjà un trigger nommer Rappel
IF OBJECT_ID ('vente.Rappel', 'TR') IS NOT NULL
   -- Alors on le supprime
   DROP TRIGGER vente.Rappel;
GO
```

```
-- On créer un trigger nommé Rappel
CREATE TRIGGER Rappel
-- Sur la table
ON vente.PRODUIT
-- Après insertion ou mise à jour de données
AFTER INSERT, UPDATE
-- On affiche un message à l'utilisateur
AS RAISERROR ('Notification', 16, 10);
GO
```

On test notre trigger. Pour cela, on met à jour le libellé (LIBART) de l'article 'T001' (CODART).

```
UPDATE vente.PRODUIT
SET LIBART = 'Un produit test'
WHERE CODART = 'T001'
```

Quelques informations sur les déclencheurs :

sys.triggers : Affiche des informations sur les déclencheurs d'étendue base.
sys.triggers_event : Affiche les évènements de base de données qui exécutent des déclencheurs.
sys.sql_modules : Affiche la définition d'un déclencheur d'étendue base.
sys.assembly_modules : Affiche des informations sur les déclencheurs CLR d'étendue BD.
sys.server.triggers : Affiche des informations sur les déclencheurs d'étendue serveur.
sys.server_triggers_event : Affiche les évènements serveur qui exécutent des déclencheurs.
sys.sql_modules : Affiche des informations sur les triggers CLR d'étendue serveur.
sys.server_assembly_modules : Affiche des informations sur les déclencheurs CLR d'étendue serveur.

Modification d'un déclencheur du DDL

Un déclencheur peut être modifié sans avoir à être supprimé, grâce à l'instruction **ALTER TRIGGER** : la définition modifiée remplacera la définition existante.

```
ALTER TRIGGER nom_déclencheur
ON table / View
[WITH ENCRYPTION]
FOR {[INSERT][,][UPDATE][,][DELETE]}
[WITH APPEND]
[NOT FOR REPLICATION]
AS
```

Un déclencheur peut être activé ou désactivé : Un déclencheur désactivé n'est pas lancé lors de l'instruction **INSERT**, **UPDATE** ou **DELETE**.

```
ALTER TABLE table
{ENABLE | DISABLE} TRIGGER
{ALL | nom_déclencheur[,…n]}
```

Exemple 1 : Nous désactivons le trigger « Supp_Ligne_Vendre » de la table VENDRE. Un déclencheur peut être activé (**ENABLE**) ou désactivé (**DISABLE**)

```
Use Papyrus
GO
```

```
ALTER TABLE vente.VENDRE
DISABLE TRIGGER Supp_Lignes_Vendre
```

Exemple 2 : Dans l'exemple suivant, nous désactivons le trigger « securite_trigger » de la base de données « Papyrus ».

```
Use Papyrus
GO
DISABLE TRIGGER securite_trigger
ON DATABASE
```

Suppression d'un déclencheur du DDL

Comme toujours, il est possible de supprimer un objet de la base de données de deux manières, nous allons montrer les deux.

Par code T-SQL

La syntaxe de suppression d'un déclencheur est simple. Elle est la suivante :

```
DROP TRIGGER trigger_log ON DATABASE
```

On utilise donc l'instruction suivie du nom unique du déclencheur. Il est par la suite nécessaire de préciser si le déclencheur DDL est défini sur une base de données ou sur le serveur entier.

Par l'interface graphique

Il vous suffit dans un premier temps de déployer les nœuds de l'explorateur d'objet de SSMS, jusqu'à votre déclencheur de cette manière :



Il vous suffit alors simplement de faire un clic droit sur le déclencheur à supprimer, par exemple **trigger_log**, et de choisir l'option « **Supprimer** ».

Déboguer le Transact SQL

SQL Server Management Studio (SSMS) dispose d'un outil de débogage Transact SQL afin de mettre au point de façon rapide et simple les procédures et fonctions.

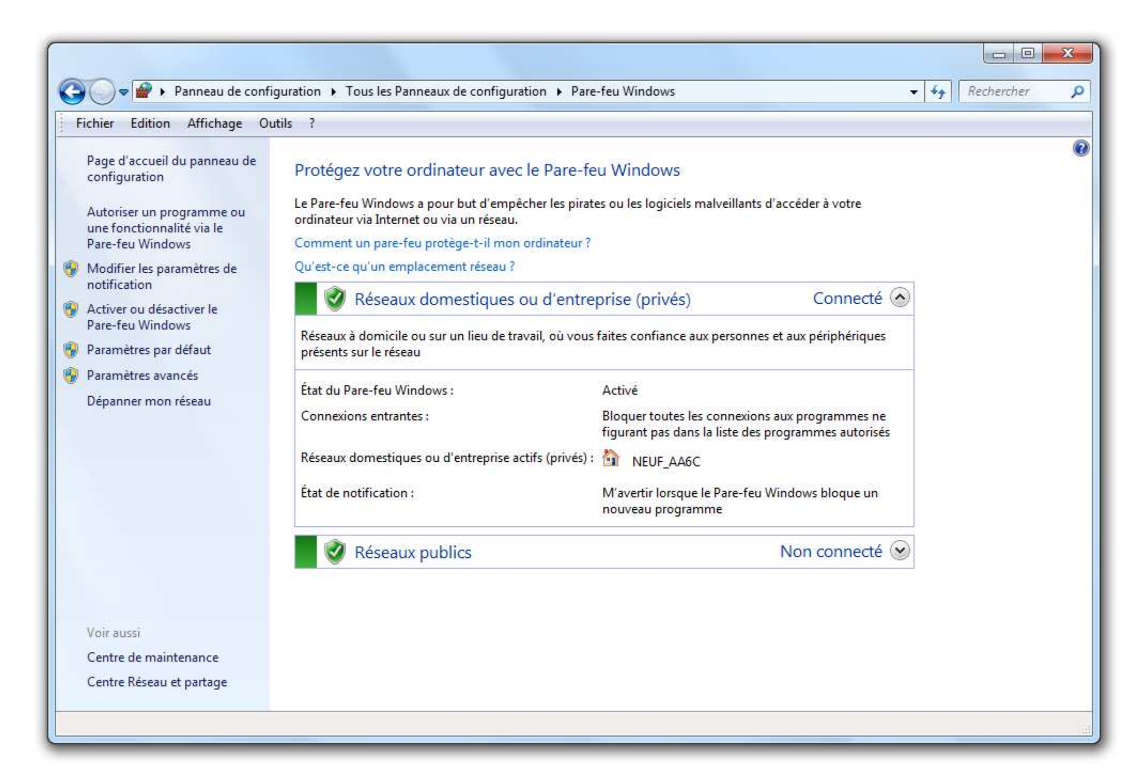
Activer le débuggeur

Le débogage Transact-SQL avec Visual Studio nécessite un serveur de base de données SQL Server et de configurer les autorisations SQL Server appropriées. Sur l'ordinateur qui exécute l'instance de SQL Server, vous devez ajouter les éléments suivants à la liste des exceptions du Pare-feu Windows :

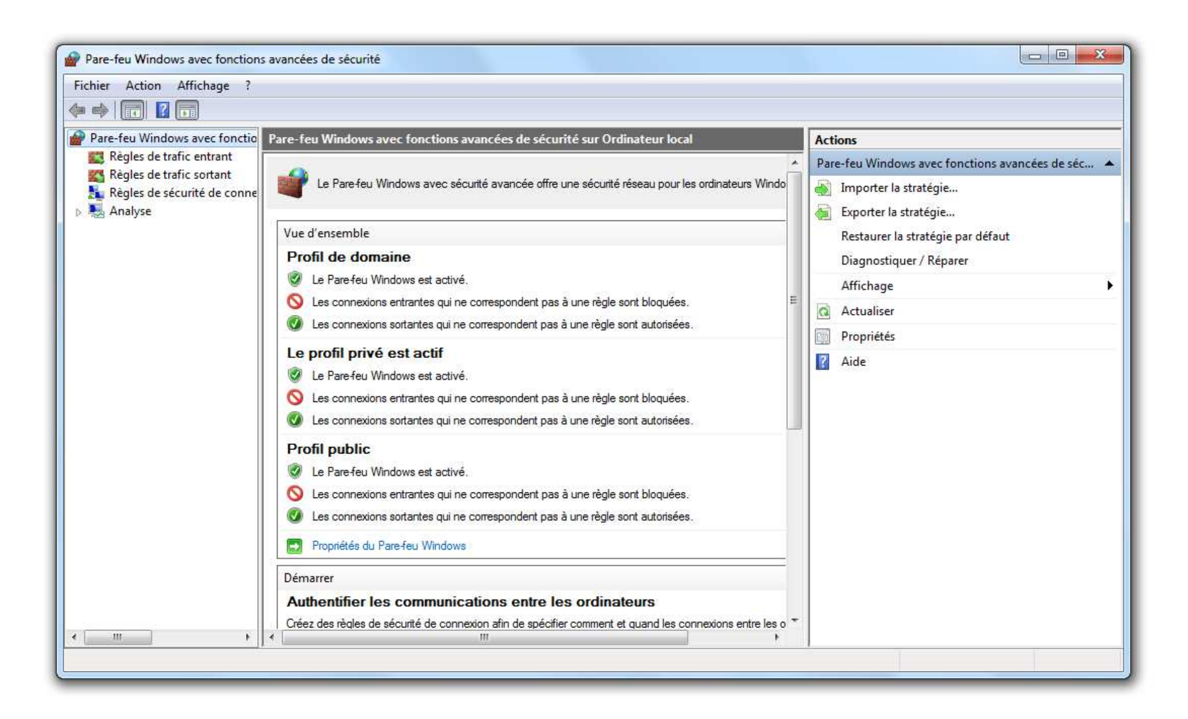
- Ouvrir le port **TCP 135**. Si votre stratégie de domaine exige que la communication réseau s'effectue par le biais du protocole IPSec, vous devez également ouvrir les ports **UDP 4500** et **UDP 500**.

- Ouvrir une règle pour le programme « **Micorosft SQL Server Management Studio** ». Par défaut, ce programme est installé dans « C:\Program Files\Microsoft SQL Server\MSSQL10.NomInstance\MSSQL\Binn », où « NomInstance » représente MSSQLSERVER pour l'instance par défaut et le nom de l'instance pour toute instance nommée.

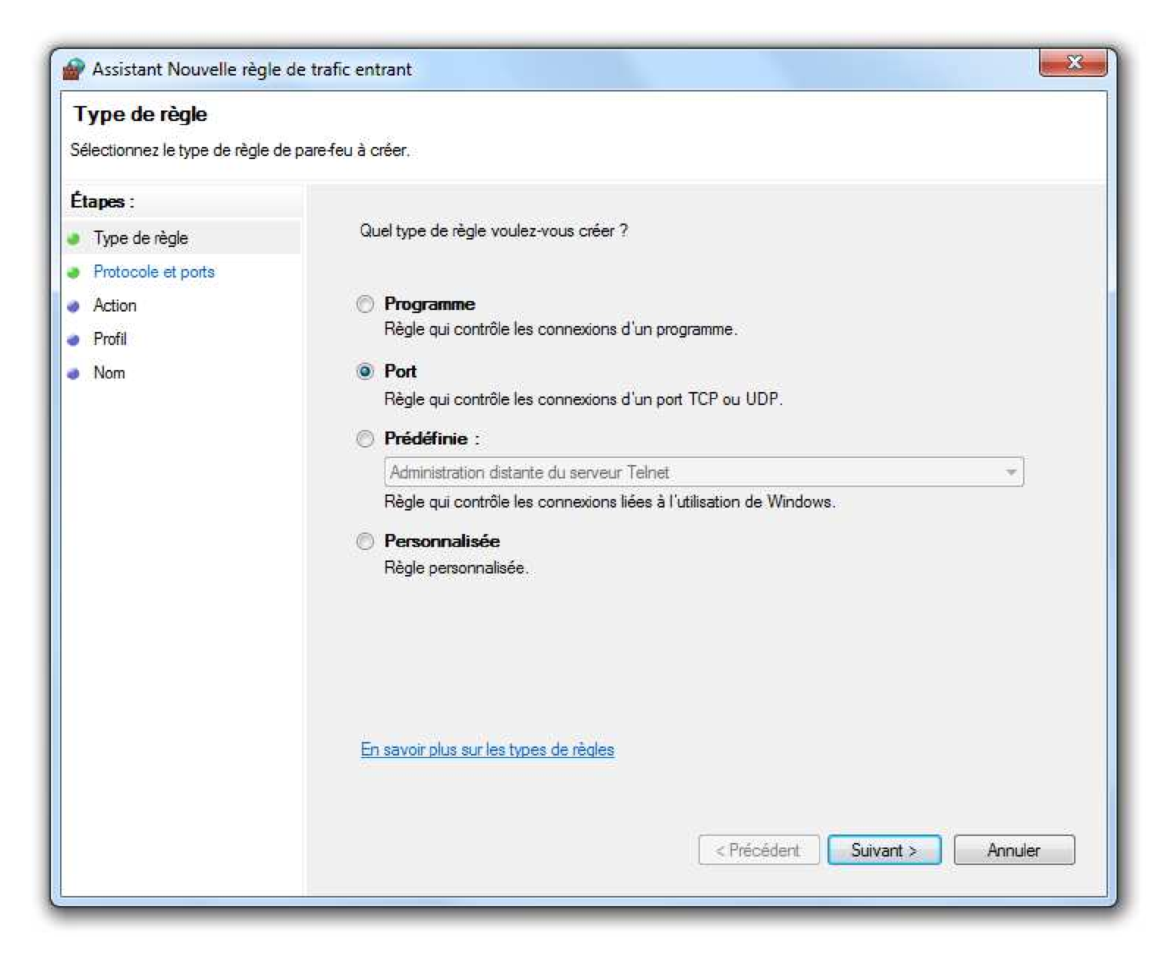
Ouvrir le « **Pare-feu Windows** » dans le panneau de configuration puis cliquez sur « **Paramètres avancés** ».



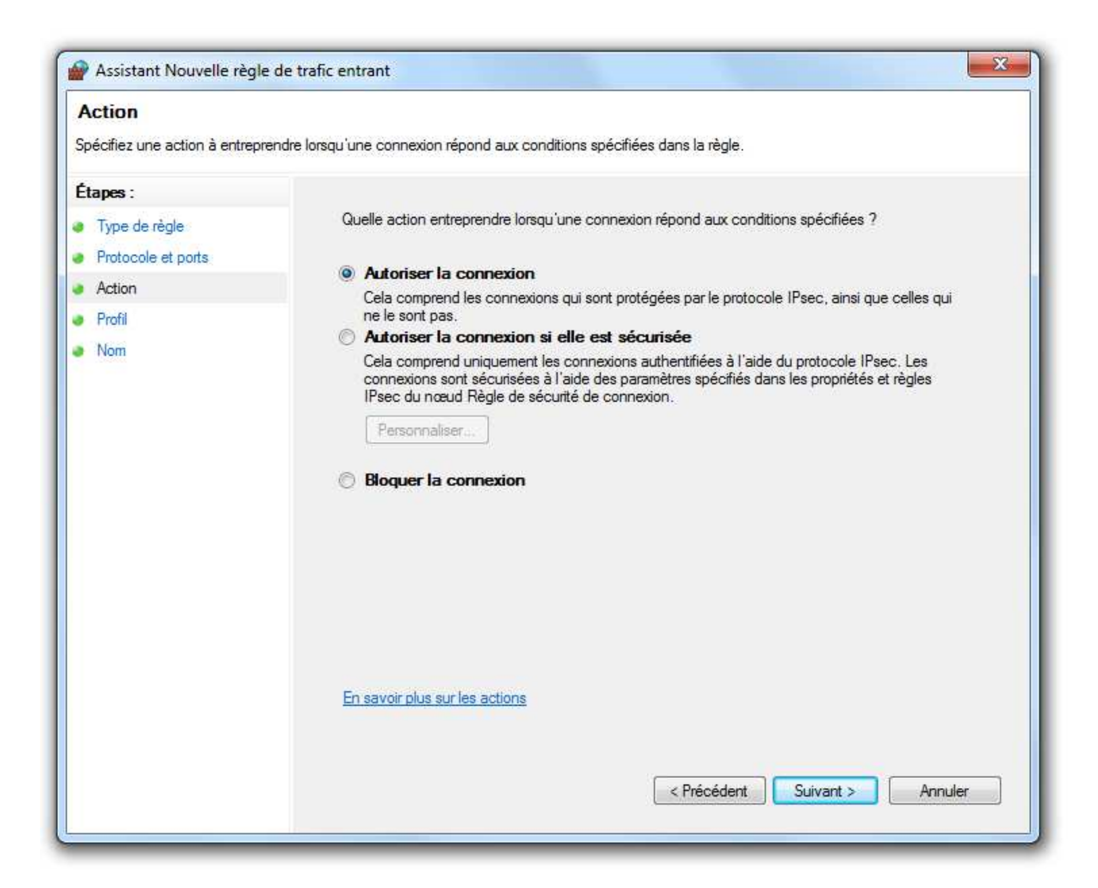
Cliquez sur « **Règles de trafic entrant** ».



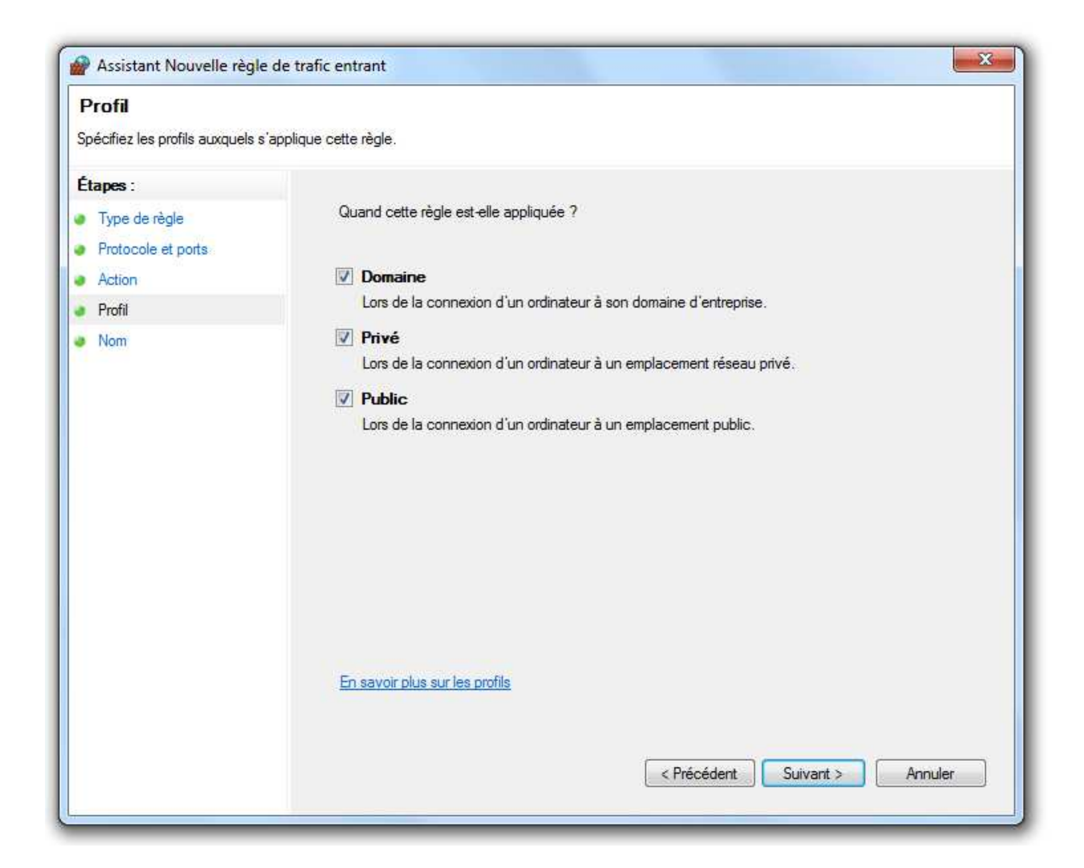
Cochez la case « **Port** » puis cliquez sur « **Suivant** ».



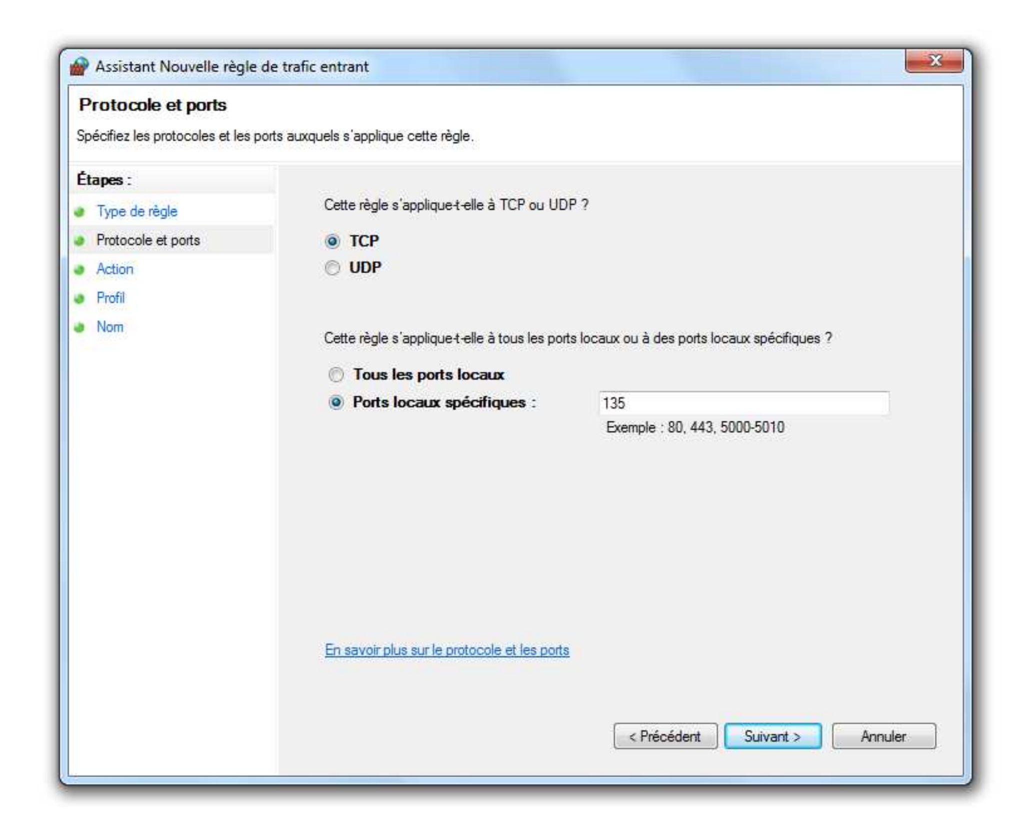
Cochez « **Autoriser la connexion** » puis « **Suivant** » :



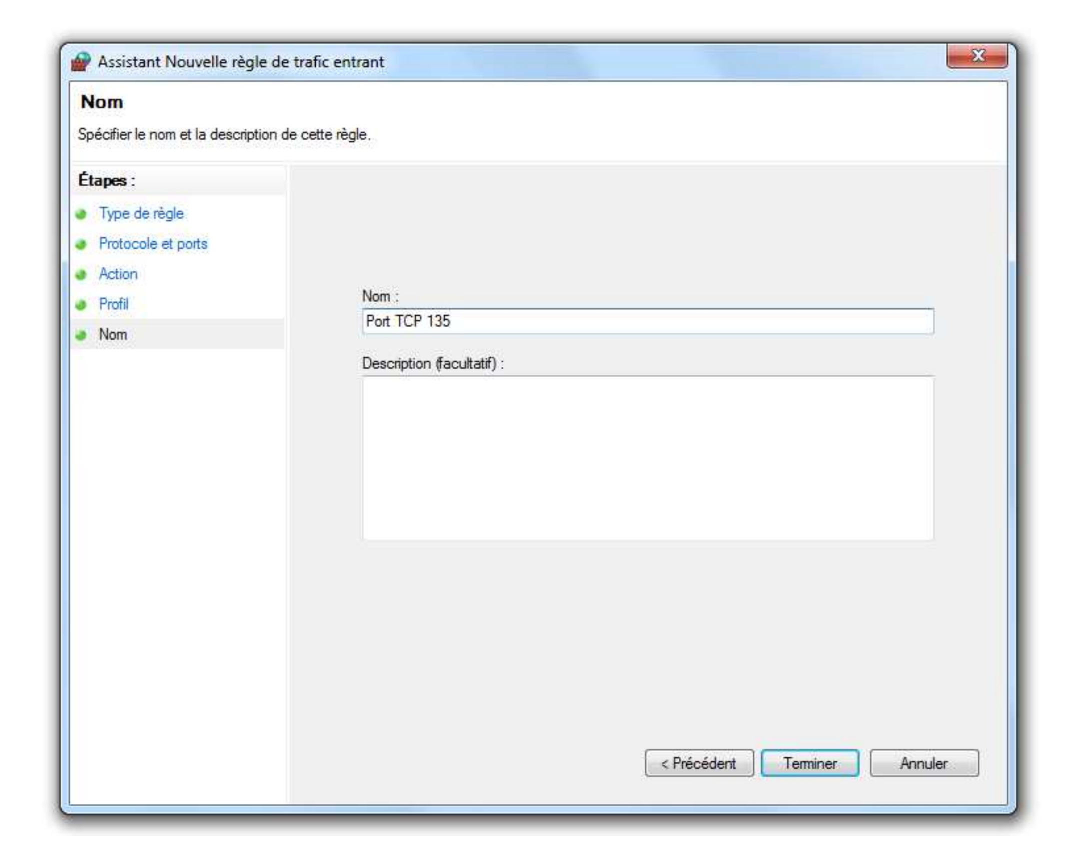
Laissez cocher les trois cases puis « **Suivant** » :



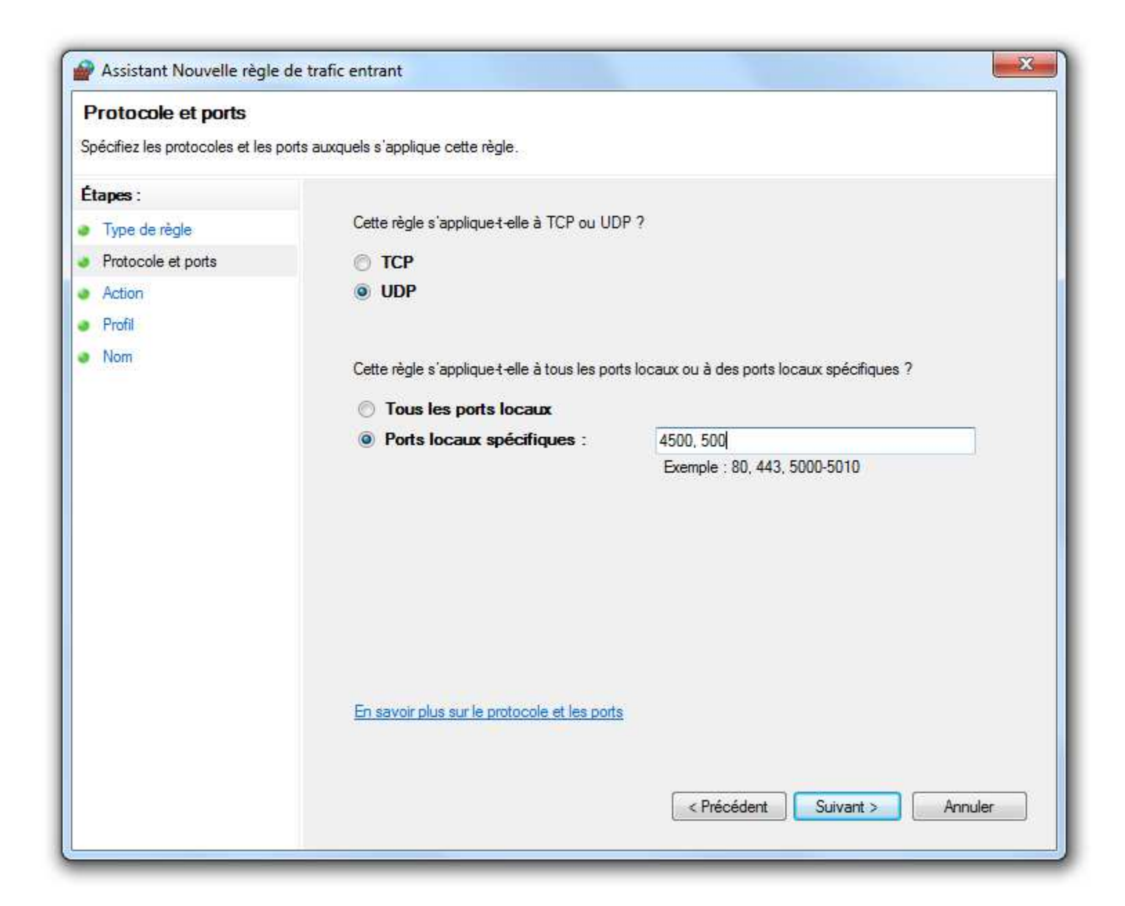
Donnez un nom à votre règle :



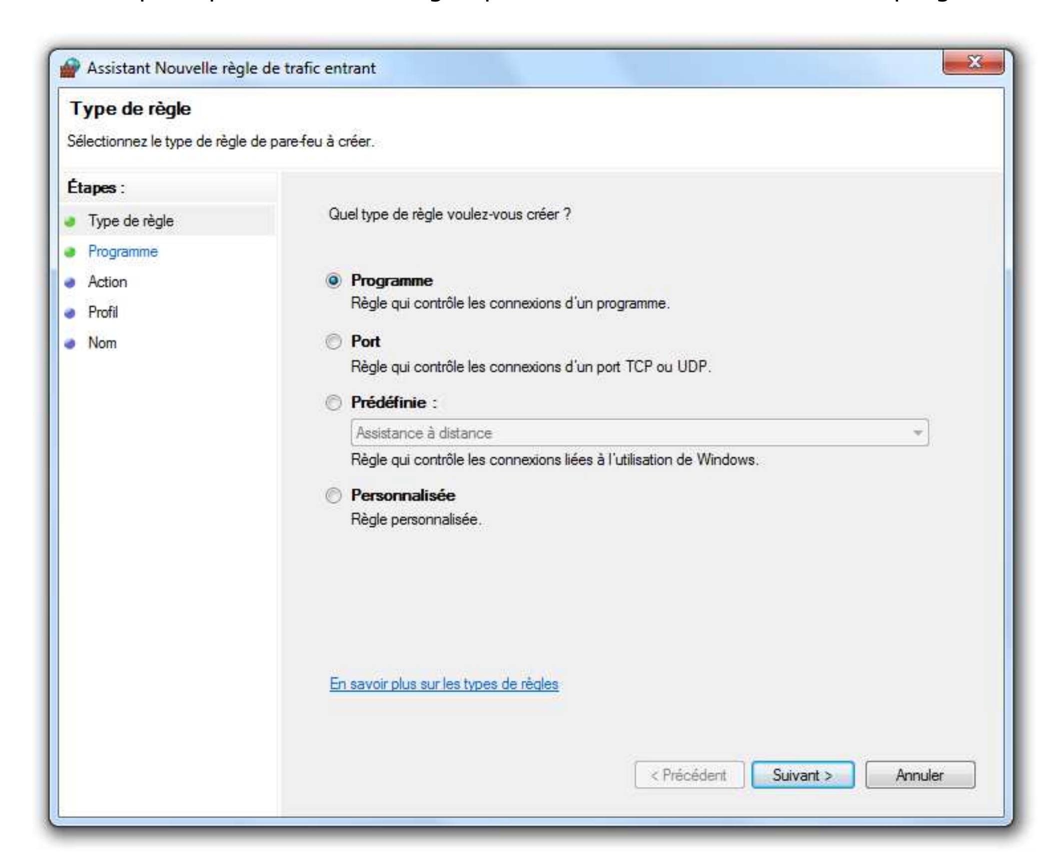
Cliquez sur « **Terminer** ».



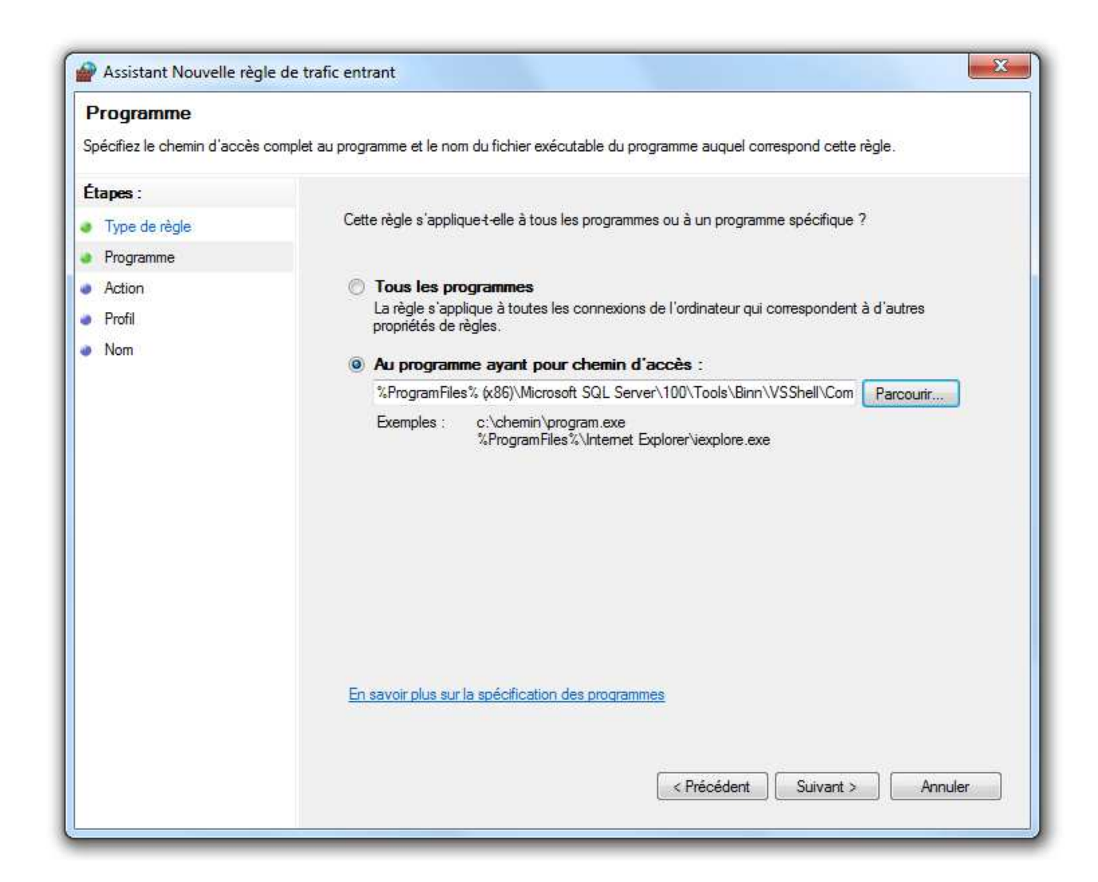
Vous devez faire la même chose pour les ports **UDP 4500** et **500**.



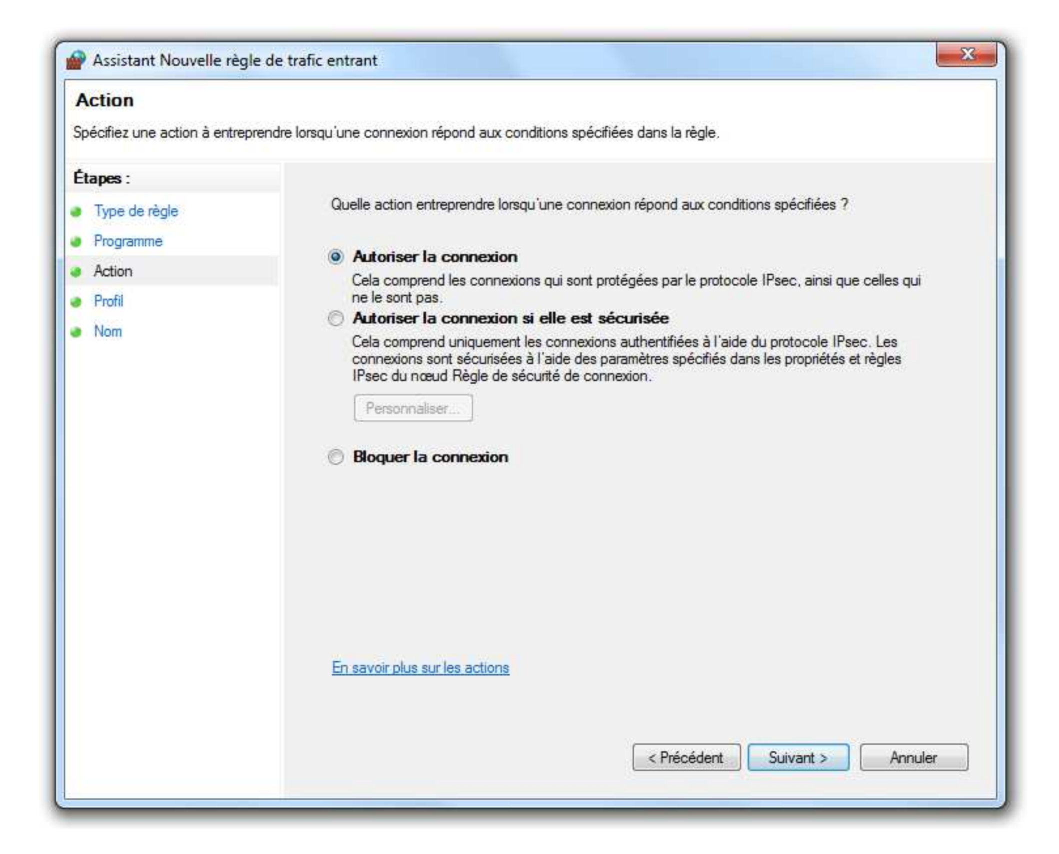
Il ne vous reste plus qu'à ouvrir une règle qui contrôle les connexions d'un programme.



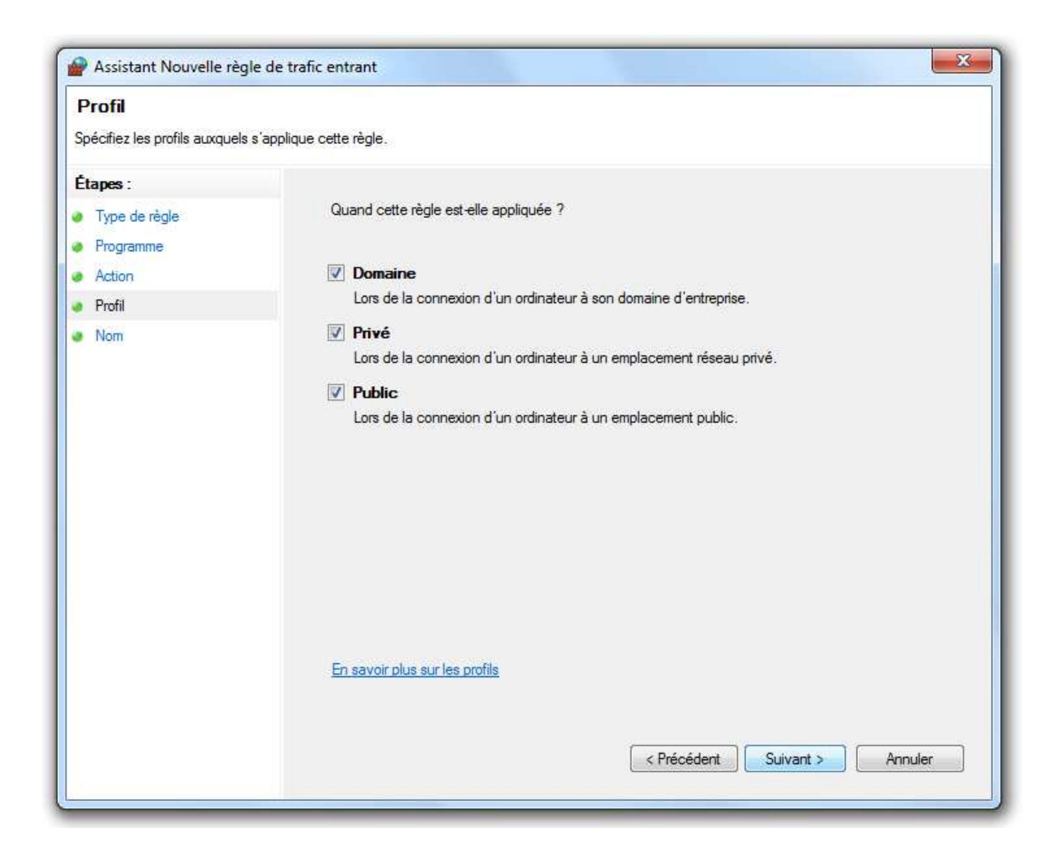
Indiquez le chemin du programme.



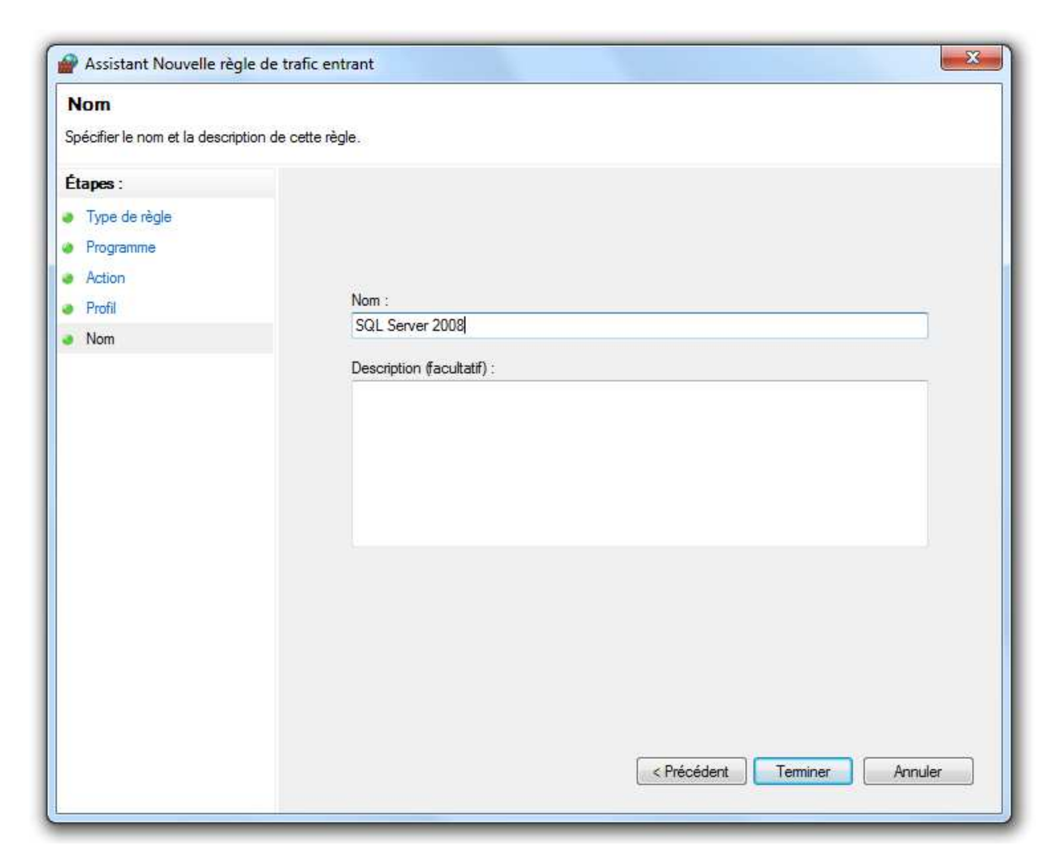
Nous laissons « **Autoritser la connexion** » puis cliquez sur « **Suivant** ».



Cliquez sur « **Suivant** ».

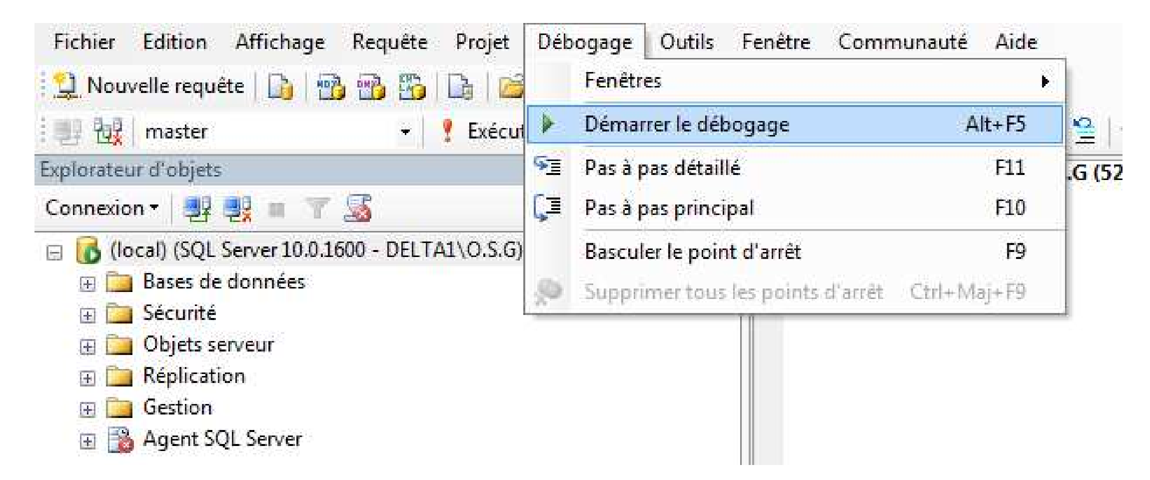


Puis donnez un nom à votre règle.



Fonctionnement du débuggeur

Voyons d'un peu plus près comment cela fonctionne. Tout d'abord lorsque l'on ouvre une fenêtre de type requête depuis SSMS, le menu général est modifié et le choix débogage apparaît.

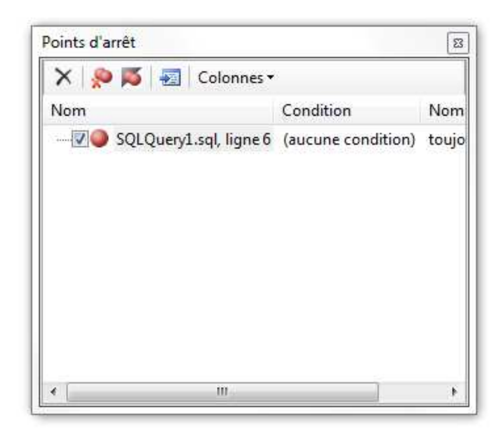


Pour explorer les possibilités offertes par cet outil, l'opération la plus simple consiste à le mettre en pratique.

Une fois le script écrit, l'ajout/suppression des points d'arrêt s'effectue en double cliquant dans la barre située à gauche de la ligne sur laquelle le point d'arrêt doit être défini. Il est également possible d'utiliser le raccourci clavier **F9**.

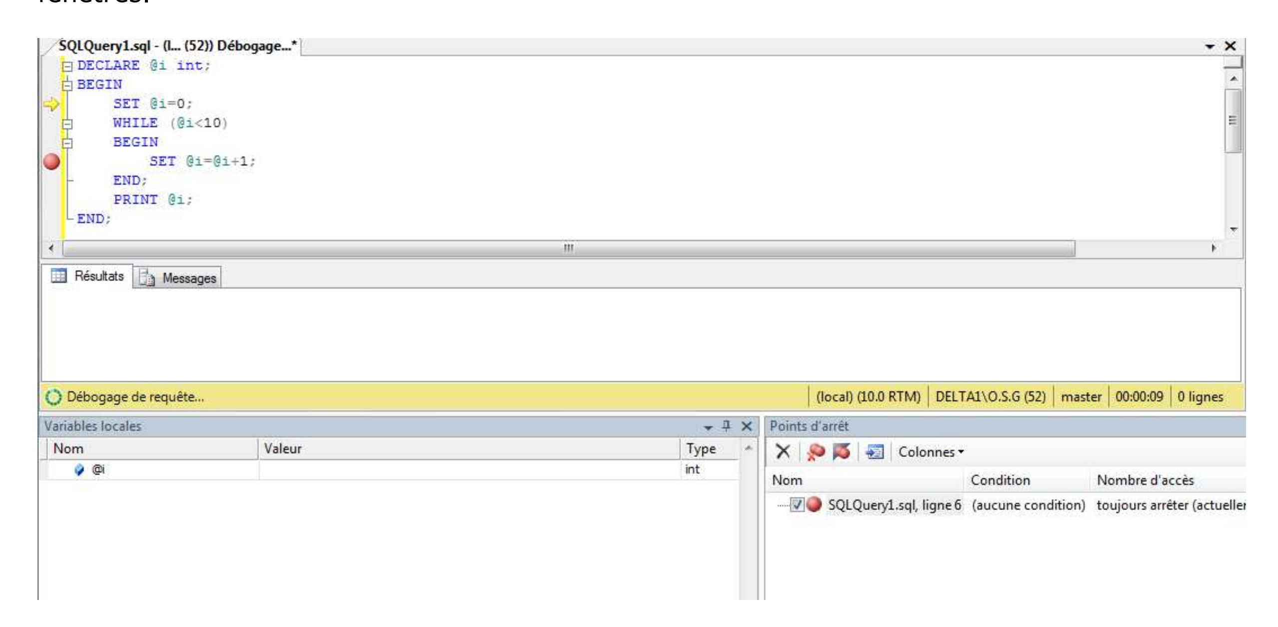
```
DECLARE @i int;
BEGIN
    SET @i=0;
    WHILE (@i<10)
    BEGIN
        SET @i=@i+1;
    END;
    PRINT @i;
END;
```

Il est possible d'avoir une vue synthétique des différents points d'arrêts qui ont été définis en demande l'affichage de la fenêtre des points d'arrêts. Cette fenêtre est accessible depuis le menu **Débogage - Fenêtres - Points d'arrêt** ou bien par le raccourci clavier **Ctrl+Alt+B**.

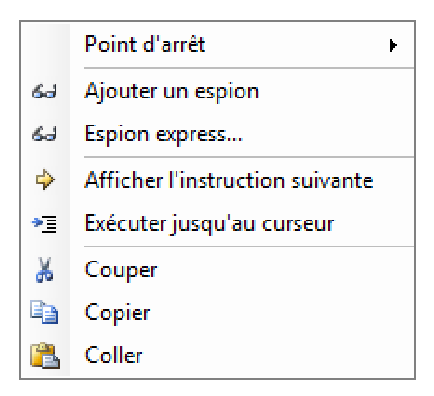


Il est également possible de supprimer un ou plusieurs points d'arrêt de façon simple à partir de cette fenêtre.

Pour lancer le script en mode débogage il ne faut bien sûr pas utiliser le raccourci clavier F5 qui permet d'exécuter un script en mode normal, mais il est nécessaire de passer par **Débogage - Démarrez le débogage** ou bien d'utiliser le raccourci clavier **Alt+F5**. Avant de lancer l'exécution du script SSMS change de perspective et donc modifie l'organisation de fenêtres.



Il est alors possible d'exécuter simplement le script en mode pas à pas, de modifier les points d'arrêts (ajouter/ supprimer/ désactiver), évaluer la valeur prise par une variable... Tous ces choix sont accessible soit de façon classique par le menu, les raccourcis clavier, le menu contextuel ou bien la barre d'outil spécifique au débogage.



Menu contextuel du mode débogage



Déboguer un déclencheur

La même technique de débogage peut être utilisée pour mettre au point un déclencheur T-SQL. La méthode est légèrement différente, car il est nécessaire de mettre en place le déclencheur (à l'aide de l'instruction **CREATE TRIGGER**) puis d'exécuter en mode débogage un script Transact SQL qui déclenche l'exécution de ce déclencheur.

Par exemple un déclencheur en mode **INSTEAD OF** est défini sur la table des catégories de la façon suivante :

```sql
use Club;
go
create trigger ioInsCat
  on categories  instead of insert as
 begin
    insert into Categories(code, libelle, anneedebut, anneefin)
      select code, UPPER(libelle),anneedebut, anneefin
      from inserted;
 end;
```

Pour exécuter ce déclencheur en mode débogage, le script de test suivant est exécuté avec un point d'arrêt sur l'instruction **INSERT** qui déclenche l'exécution du trigger.

```sql
create trigger ioInsCat
  on categories  instead of insert as
 begin
    insert into Categories(code, libelle, anneedebut, anneefin)
      select code, UPPER(libelle),anneedebut, anneefin
      from inserted;
 end;
 -- test du déclencheur
 insert into categories(code, libelle, anneedebut, anneefin)
   values ('c1','categorie1',2000, 2001),
   ('c2','categorie1',2001, 2002);
```

Arrivé au niveau du point d'arrêt il faut alors demander une exécution en **mode pas à pas détaillé** ou **F11**. Le déboguer permet alors d'explorer en mode pas à pas l'exécution du déclencheur.

Utilisation de l'envoi d'email via le protocole SMTP

Microsoft SQL Server propose un utilitaire d'envoi de mail (Database Mail). Pour l'utiliser, vous devez tout d'abord activer le service Database Mail.

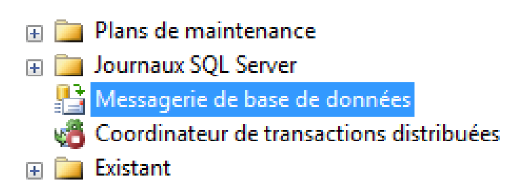
La messagerie de base de données n'est pas activée lors de l'installation de SQL Server. Utilisez l'Assistant Configuration de la messagerie de base de données pour activer et installer les objets de messagerie de base de données. Avant cela, vous devez configurer les options avancées. Pour configurer une option avancée, vous devez tout d'abord exécuter **sp_configure** après avoir attribué la valeur 1 à l'option 'show advanced options', puis vous devez exécuter **RECONFIGURE**, tel qu'indiqué dans l'exemple suivant.

```sql
-- On configure les options avancées
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
-- On active le service Database Mail XP's
EXEC sp_configure 'Database Mail XPs', '1'
GO
RECONFIGURE WITH OVERRIDE;
GO
-- On démarre le service
EXEC msdb.dbo.sysmail_start_sp;
```

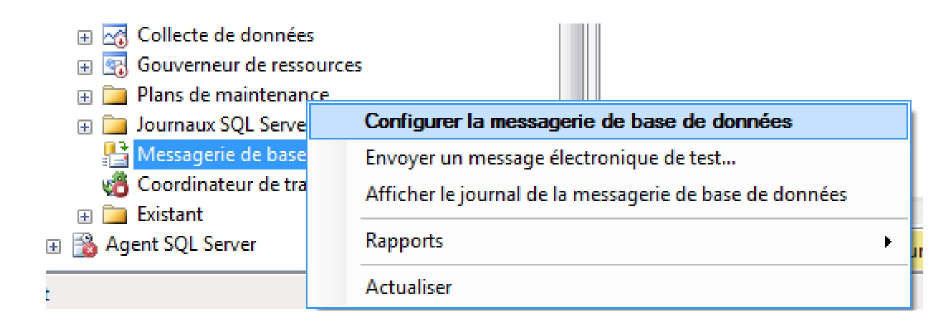
Après cette étape, Database Mail est activé pour notre serveur et nous pouvons passer à la configuration.

Par l'interface

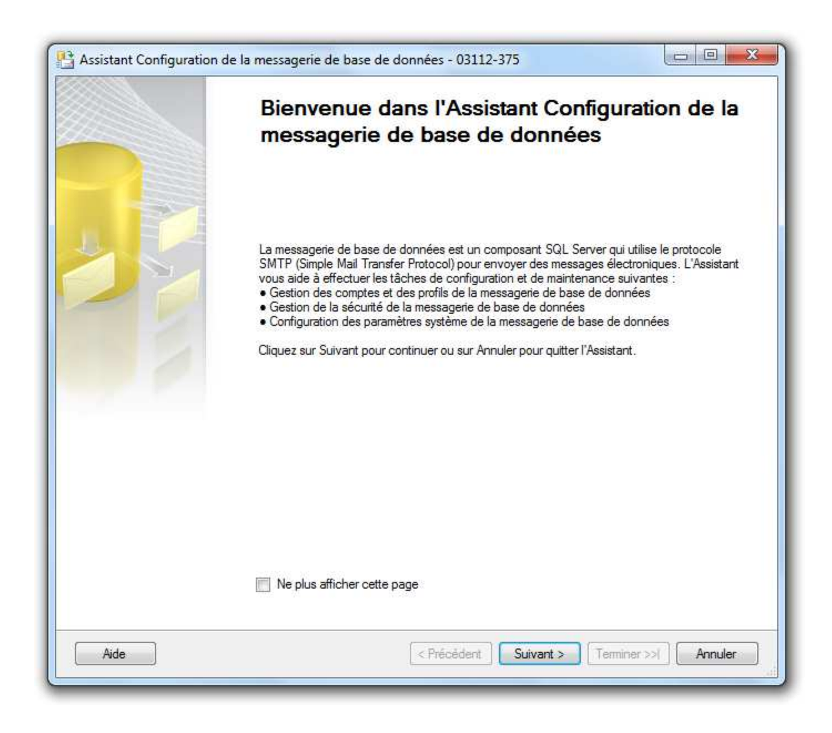
Cliquez droit sur « **Messagerie de base de données** ».



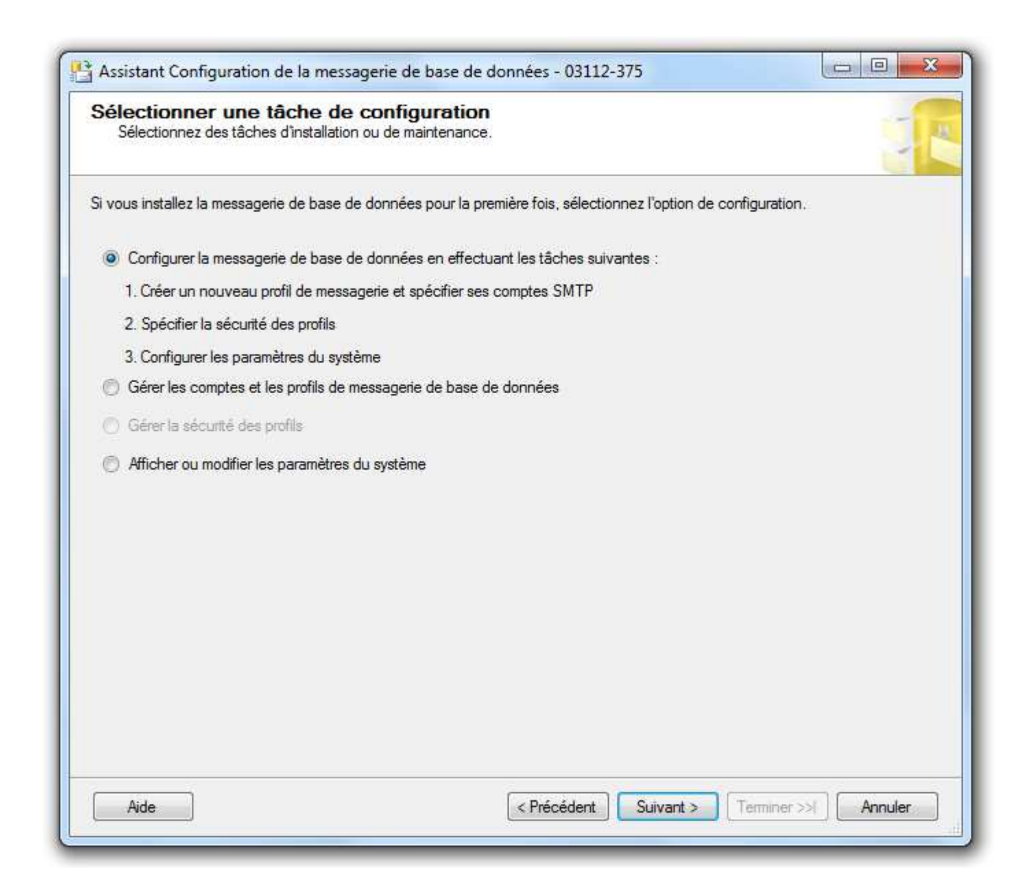
Puis « **Configurer la messagerie de base de données** ».



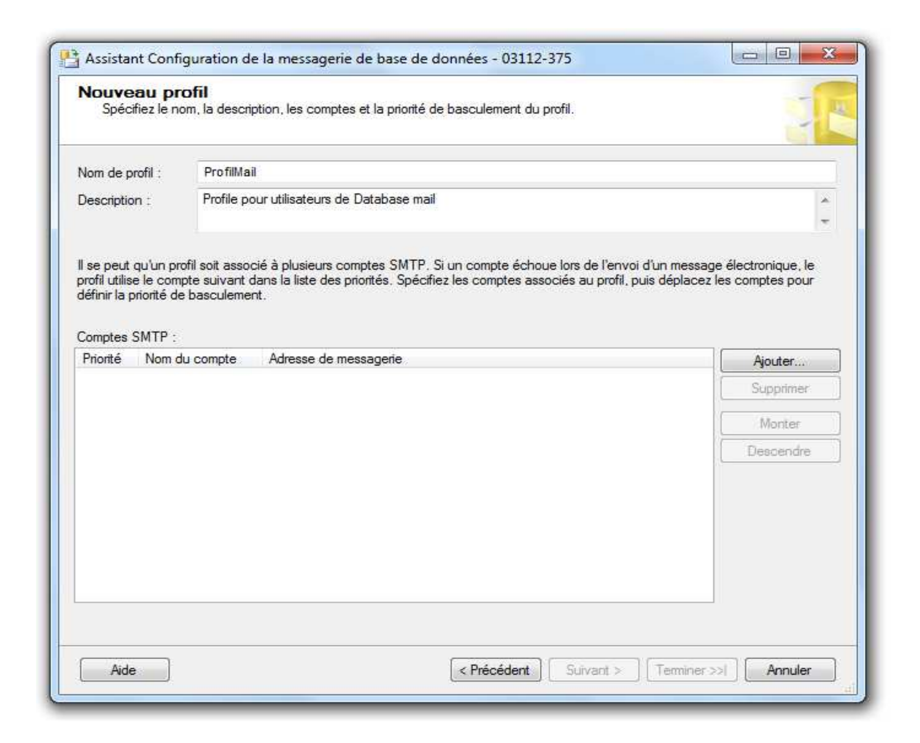
L'assistant de configuration de la messagerie de base de données s'ouvre. Cliquez sur « **Suivant** ».



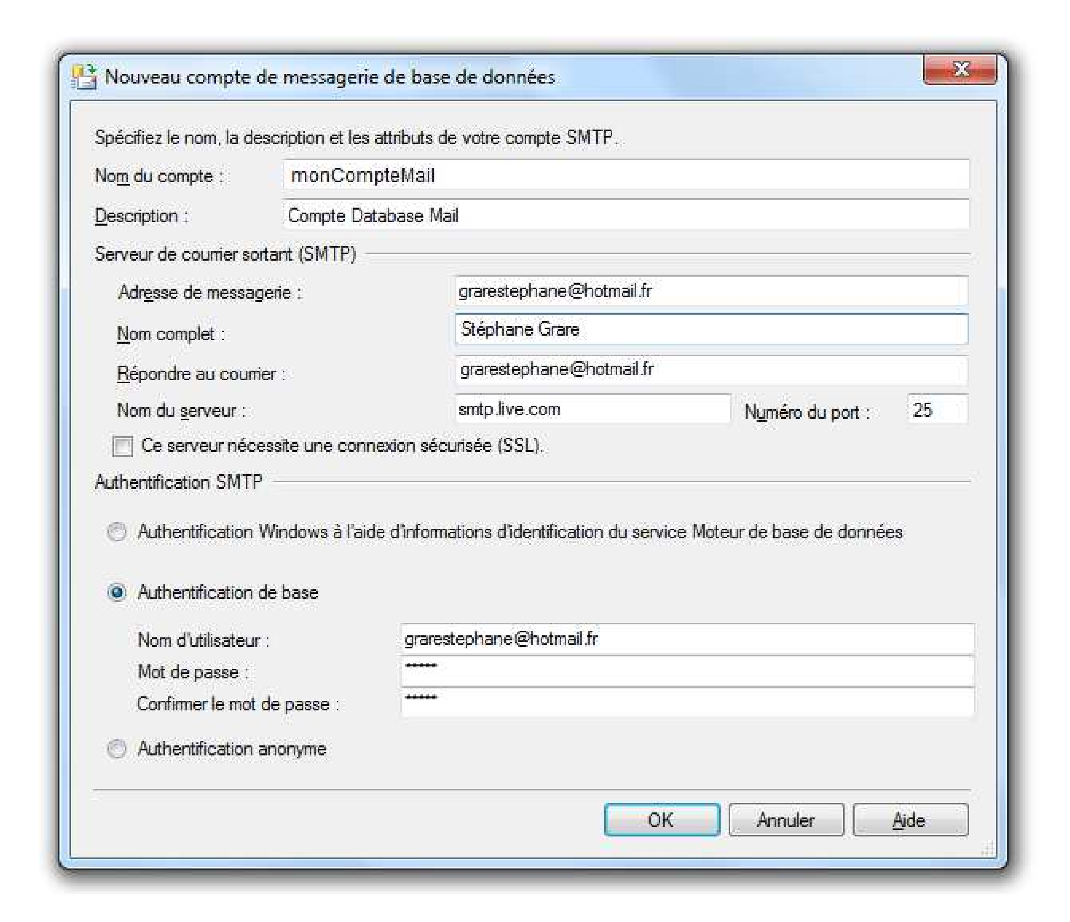
L'écran suivant nous demande ce que nous souhaitons faire maintenant, nous choisirons la première possibilité « **Configurer la messagerie de base de données**... ».



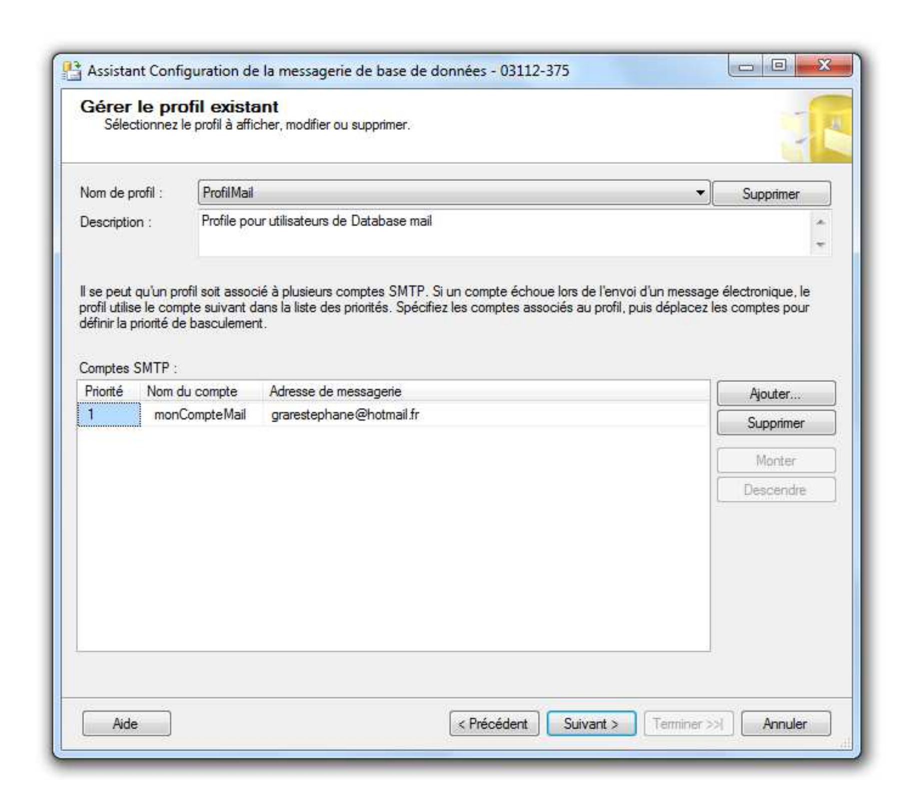
On déclare un nom et une description de notre profil.



Puis cliquez sur « **Ajouter** », afin de définir les paramètres nécessaires à l'envoi des emails. Dans cet écran, il nous faut définir les paramètres de l'adresse email que le destinataire verra comme Emetteur, mais surtout l'adresse et le port du server SMTP relais. Dans le cas où votre relais exige une authentification, vous devez le spécifier ici.

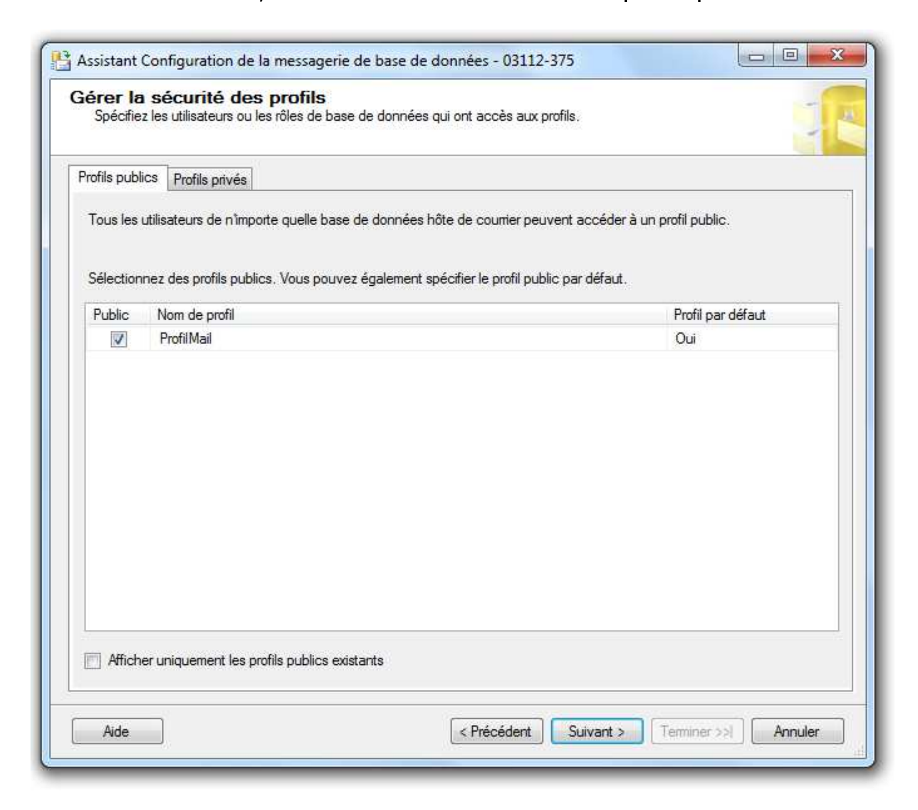


Cliquez sur « **Ok** » puis sur « **Suivant** ».

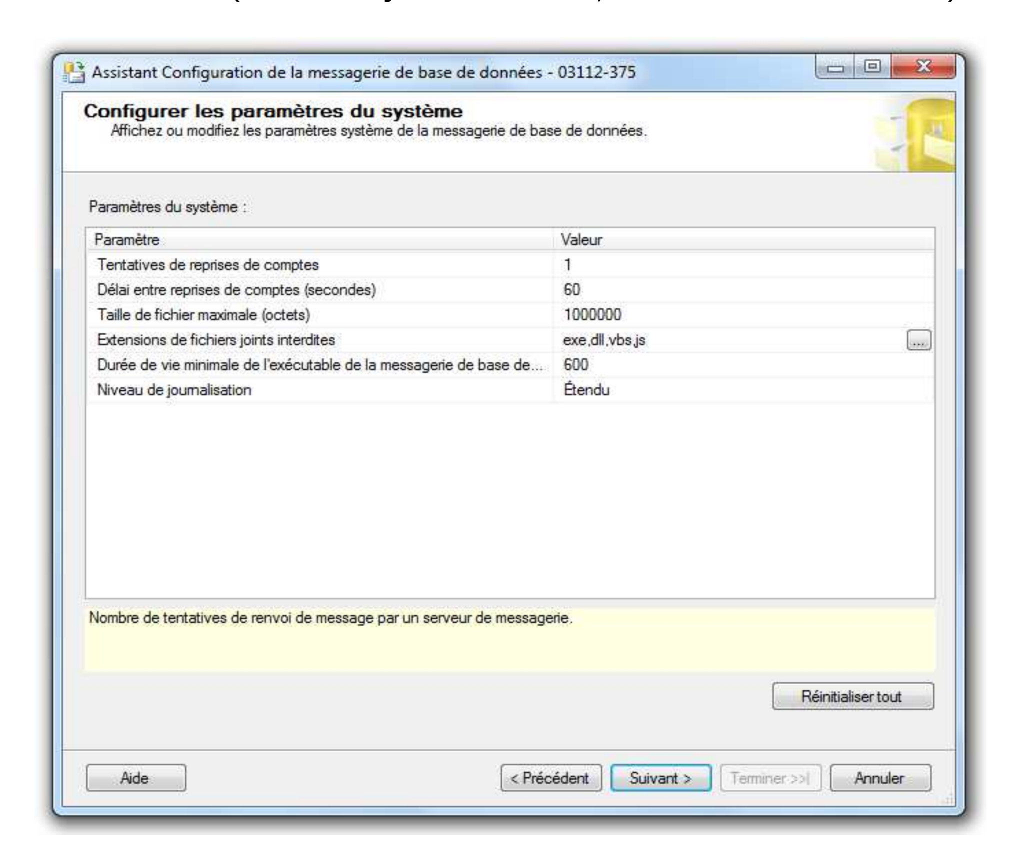


Maintenant, on doit spécifier si le profil SMTP que l'on vient de créer est public ou privé. Le fait d'être public permet que ce profil soit utilisable depuis toutes les bases de données hôtes

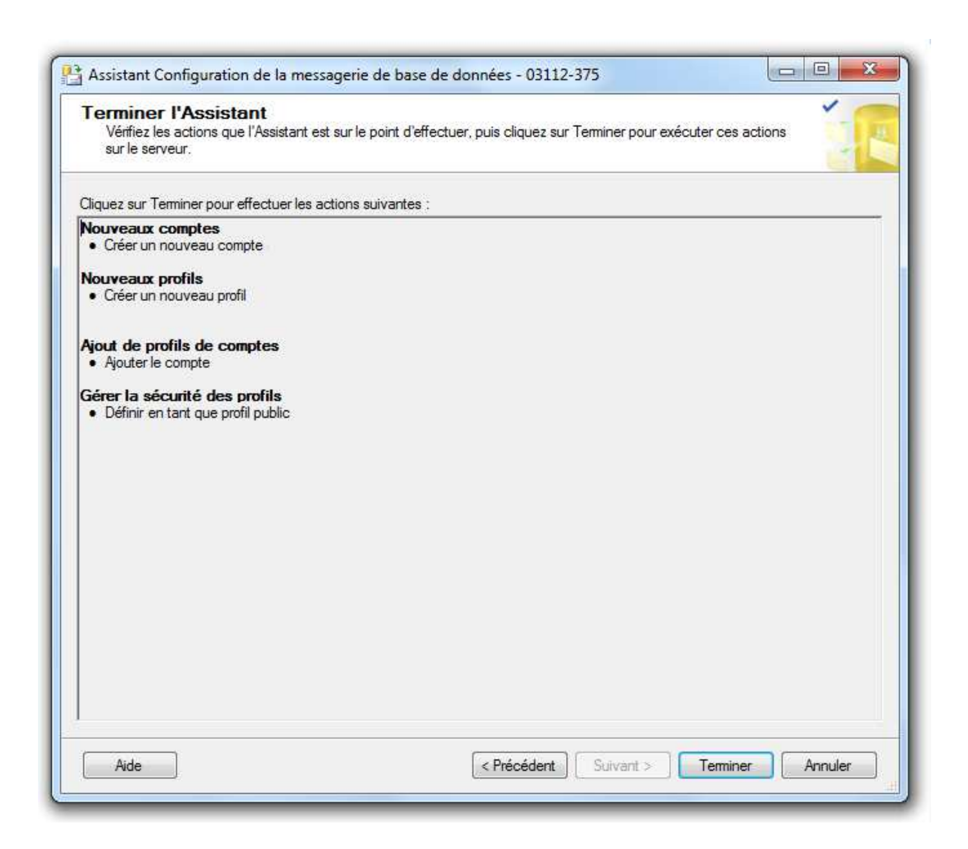
du courrier. Dans le cas d'un profil privé, le profil ne sera utilisable que par l'utilisateur que l'on aura choisi. Dans notre cas, nous souhaitons rendre ce profil public.



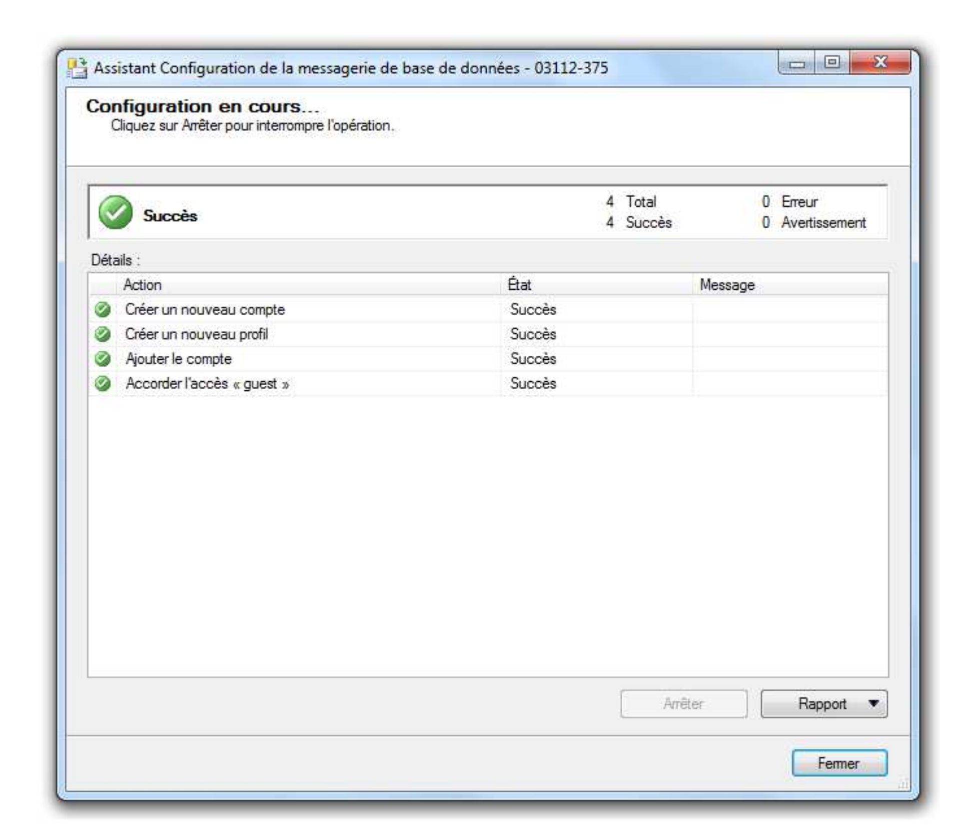
L'écran suivant nous donne un résumé des paramètres à suivre par le serveur lors d'une tentative d'envoi de mail (les fichiers joints à interdire, le nombre de tentatives...).



On a alors un récapitulatif de la configuration sélectionnée.

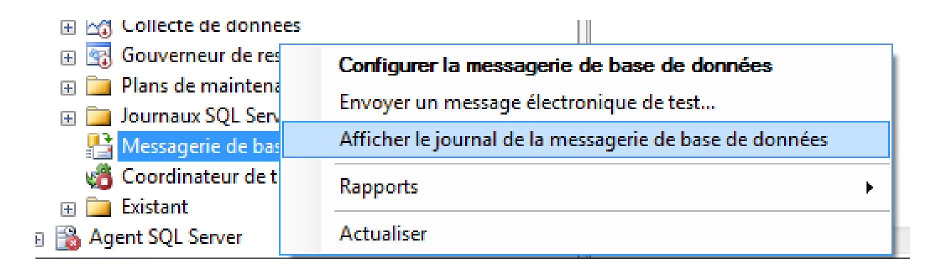


Puis le rapport d'exécution de cette configuration.

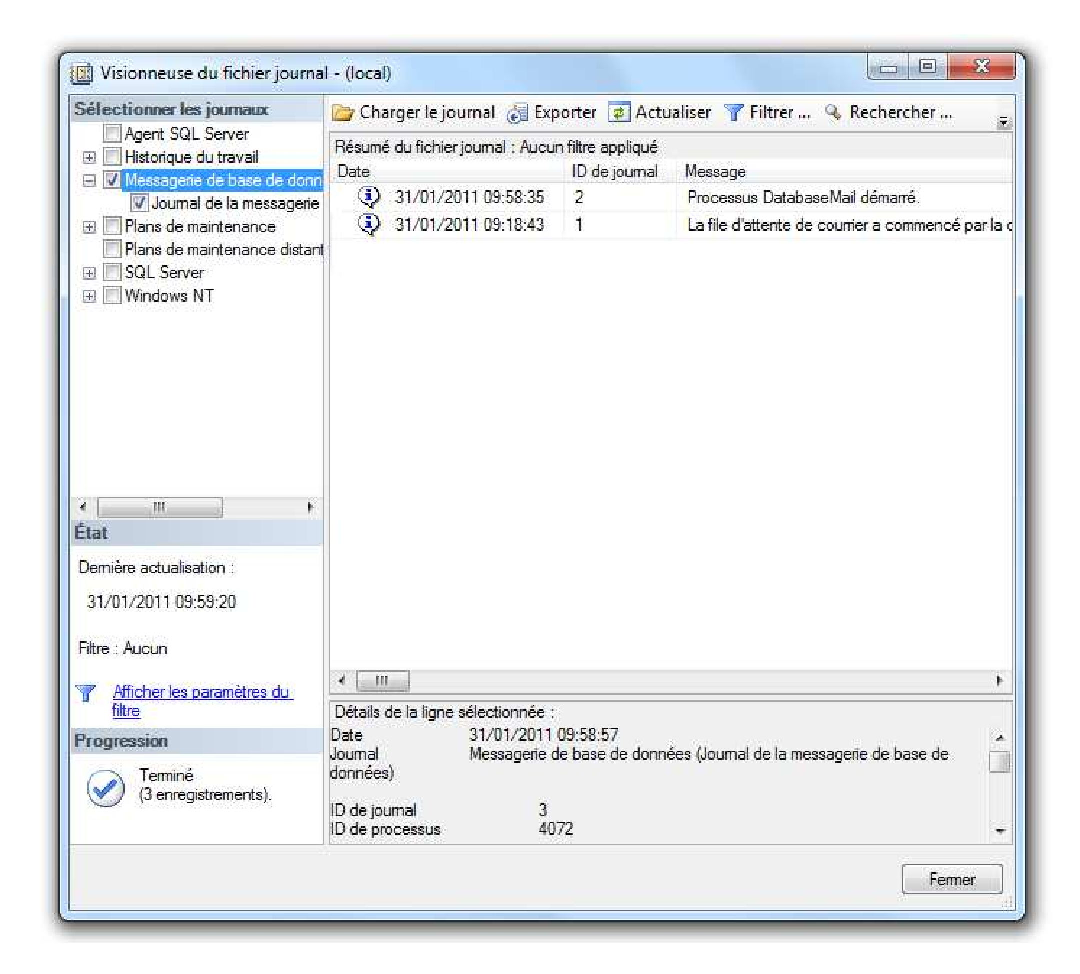


Maintenant que cette configuration est appliquée, nous pouvons effectuer un test d'envoi en cliquant avec le bouton droit sur « **Messagerie de base de données** » et sélectionnant « **Envoyer un message électronique de test** ».

Une fois cette configuration effectuée et testée, vous pouvez visionner le journal de cette messagerie, en cliquant avec le bouton droit de la souris sur « **Messagerie de base de données** » et sélectionnant « **Afficher le journal de la messagerie de la base de données** ».



Vous aurez alors l'écran suivant qui reprend les journaux (comme ceux de Windows).



Par le code

Quatre étapes sont nécessaires : créer un compte utilisateur, créer un profil, ajouter le compte au profil et choisir les utilisateurs de msdb qui pourront accéder au profil (public, privé). Ces étapes peuvent se faire par script de la manière suivante :

- <u>Créer un compte utilisateur</u> :

```
EXECUTE msdb.dbo.sysmail_add_account_sp
```

```
  @account_name = 'monCompteMail',
  @description = 'Compte Database Mail',
  @email_address = 'grarestephane@hotmail.fr',
  @replyto_address = 'grarestephane@hotmail.fr',
  @display_name = 'Stéphane Grare',
  @username='grarestephane@hotmail.fr',
  @password='monmotdepasse',
  @mailserver_name = 'smtp.live.com',
  @port=25
```

- Créer un profil de comptes mail :

```
EXECUTE msdb.dbo.sysmail_add_profile_sp
  @profile_name = 'ProfilMail',
  @description = 'Profile pour utilisateurs de Database mail'
```

- Assigner le compte au profil : cela se fait avec la procédure stockée **msdb.dbo.sp_sysmail_add_profileaccount**. Il vous est possible de voir les paramètres de chaque procédure stockée via le navigateur d'objets de Management Studio :

```
EXECUTE msdb.dbo.sysmail_add_profileaccount_sp
  @profile_name = 'ProfilMail',
  @account_name = 'monCompteMail',
  -- Désigne l'ordre dans lequel sont triés les comptes dans le profil
  @sequence_number = 1
```

- Assigner le droit d'utilisation du profil à des utilisateurs : Il faut que les utilisateurs puissent utiliser le profil. La procédure **sysmail_add_principalprofil** est là pour cela :

```
  EXECUTE msdb.dbo.sysmail_add_principalprofile_sp
  @profile_name = 'ProfilMail',
  @principal_name = 'public',
  @is_default = 1;
```

Ici, nous disons que le profil que nous avons créé peut être utilisé par les utilisateurs appartenant au rôle 'public' pour la base de données MSDB. Nous aurions très bien pu spécifier un utilisateur particulier.

- Envoyer un mail : Exécutons le script suivant.

```
EXEC msdb.dbo.sp_send_dbmail @recipients='grarestephane@hotmail.fr',
  @subject = 'Test',
  @body = 'MON PREMIER MESSAGE !!!!',
  @body_format = 'HTML'
```

Si tout se passe bien, vous recevrez votre mail avec le message attendu. N'oubliez pas de libérer le port 25 pour l'envoi de mail depuis votre serveur, il arrive que le serveur de messagerie bloque tout simplement le mail...

Quelle que soit la possibilité choisie (script ou visuel), votre serveur SQL est maintenant capable d'envoyer des emails. Nous allons maintenant voir comment utiliser cette possibilité à travers un exemple.

Exemple : Utilisation d'un déclencheur DML avec un message de rappel par courrier électronique. L'exemple suivant envoie un message électronique à une personne spécifiée (Stéphane Grare) lorsque la table PRODUIT est modifiée.

```
USE Papyrus;
GO
-- S'il existe déjà un trigger nommer Rappel
IF OBJECT_ID ('vente.RappelMail', 'TR') IS NOT NULL
   -- Alors on le supprime
   DROP TRIGGER vente.RappelMail;
GO
CREATE TRIGGER RappelMail
ON vente.PRODUIT
AFTER INSERT, UPDATE, DELETE
AS
   EXEC msdb.dbo.sp_send_dbmail
        @profile_name = 'ProfilMail',
        @recipients = 'grarestephane@hotmail.fr',
        @body = 'Un message de votre trigger Rappel.',
        @subject = 'Trigger Rappel';
GO
```

On test notre trigger :

```
USE Papyrus
UPDATE vente.PRODUIT
SET STKPHY = STKPHY -1
WHERE CODART = 'Z002'
```

```
USE Papyrus
UPDATE vente.PRODUIT
SET STKPHY = STKPHY -1
WHERE CODART = 'Z002'
```

Messages

```
(1 ligne(s) affectée(s))
Courrier en file d'attente.

(1 ligne(s) affectée(s))
```

Exécution de requête réussie.

Comment obtenir la liste des tables d'une base de données ?

Vous avez beaucoup de possibilités pour connaître la liste des tables d'une base de données. Nous vous recommandons d'utiliser les vues d'informations de schéma.

```
Use Papyrus
GO
SELECT table_name
FROM information_schema.tables
WHERE table_type='BASE TABLE'
```

Vous pouvez aussi passer par la procédure stockée **sp_tables** ou encore passez par les tables systèmes.

```
Use Papyrus
GO
SELECT name FROM sysobjects WHERE type='U'
```

Comment connaître la liste des colonnes d'une table ?

Comme pour la liste des bases de données d'un serveur, SQL Server offre trois possibilités :

1 - La consultation des vues d'informations de schéma.

```
Use Papyrus
GO
SELECT COLUMN_NAME, ORDINAL_POSITION
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME='PRODUIT'
```

2 - L'utilisation de la procédure stockée **sp_columns**

```
Use Papyrus
GO
EXEC sp_columns 'PRODUIT'
```

3 - L'utilisation de la procédure stockée sp_help

```
Use Papyrus
GO
EXEC sp_help 'vente.PRODUIT'
```

Comment lister l'ensemble des vues d'une base de données SQL Server ?

La liste des vues d'une base de données de SQL-Server est accessible grâce à une requête sur les tables systèmes : **sysobjects**, **syscomments** et **sysusers**.

```sql
SELECT name
  FROM sysobjects
 WHERE type='V'
```

Mais il est recommandé d'utiliser les vues d'informations de schemas.

```sql
SELECT *
  FROM information_schema.views
```

Comment lister l'ensemble des UDF d'une base de données SQL Server ?

La liste des fonctions définies par l'utilisateur de SQL-Server est accessible grâce à une requête sur les tables systèmes : **sysobjects**, **syscomments** et **sysusers**.

```sql
SELECT name
  FROM sysobjects
 WHERE type='FN'
```

Comment lister l'ensemble des procédures stockées d'une base de données SQL Server ?

La liste des procédures stockées de SQL-Server est accessible grâce à une requête sur les tables systèmes : **sysobjects**, **syscomments** et **sysusers**.

```sql
SELECT name
  FROM sysobjects
 WHERE type='P'
```

On peut également utiliser la méthode des vues d'informations de schéma :

```sql
SELECT *
FROM INFORMATION_SCHEMA.ROUTINES
```

Ou encore, utiliser la procédure stockée : **sp_stored_procedures**

Comment lister l'ensemble des déclencheurs d'une base de données SQL Server ?

La liste des triggers de SQL-Server est accessible grâce à une requête sur les tables systèmes : **sysobjects**, **syscomments** et **sysusers**.

```sql
SELECT
    o.name,  o.xtype,  c.text,    u.name,    o.crdate
FROM
    dbo.sysobjects o
INNER JOIN dbo.syscomments c
    ON c.id = o.id
INNER JOIN dbo.sysusers u
    ON u.uid = c.uid
WHERE
    xtype = 'TR'
```

Quelle est la requête qui permet de savoir quelles colonnes d'une table servent de clé primaire ?

Il existe une procédure stockée pour cela :

```
EXEC sp_pkeys @table_name='PRODUIT'
```

Quelle commande permet d'afficher la description d'une table sous SQLServer ?

```
sp_help 'vente.PRODUIT'
```

Ou

```
SELECT
 column_name AS champ,
COALESCE(domain_name,
cast(data_type as varchar(128))+ ISNULL(' ' + cast(character_maximum_length
as varchar(10)) ,'')) as type_donnee,
CASE UPPER(IS_NULLABLE)
   when 'YES' then ''
   when 'NO' then 'Oui'
   when Null then ''
   else  IS_NULLABLE
END as Obligatoire,
'' as description
FROM INFORMATION_SCHEMA.columns
WHERE
 table_name  = 'PRODUIT'
ORDER BY table_name, ordinal_position
```

Comment récupérer la valeur par défaut d'un champ d'une table ?

```
 SELECT cdefault
FROM syscolumns
WHERE id = object_id('PRODUIT')
and name = 'LIBART'
```

Quel est le nombre de lignes de chacune des tables d'une base de données ?

```
SELECT O.Name AS Table_Name, I.Rows AS Rows_Count
FROM sysobjects O join sysindexes I
          ON O.id=I.id
WHERE O.xtype='U'
```

Comment connaître le nom de la base de données en cours ?

Pour connaître le nom de la base de données en cours, vous pouvez utiliser la fonction **DB_NAME().**

```
SELECT  DB_NAME() AS BASE_DE_DONNEES_EN_COURS
```

Comment afficher la liste des bases de données d'un serveur ?

Vous avez trois méthodes au choix :

1- L'utilisation des vues d'informations de schéma, Exemple :

```
SELECT CATALOG_NAME
FROM INFORMATION_SCHEMA.SCHEMATA
Go
```

2 - La consultation des tables système, bien que non recommandée pour des raisons de portabilité Exemple :

```
USE master
Go
SELECT name as BaseDedonneeDuServeur
FROM sysdatabases
Go
```

3 - L'utilisation de la procédure stockée **sp_databases** Exemple:

```
EXEC sp_databases
go
```

Comment changer le type de données d'une colonne ?

Pour changer le type de données d'une colonne, MS SQL Serveur fournit la clause **Alter Column**. Cet exemple ferait l'affaire:

```
ALTER TABLE MyTable
ALTER COLUMN MyColumn NVARCHAR(20) NOT NULL
```

Vous pouvez également procéder comme ceci :

- Démarrer une transaction sérialisée;
- Créer une nouvelle table avec le nouveau type de données tel que souhaitée;
- Importer les données de l'ancienne table vers la nouvelle;
- Supprimer l'ancienne table;
- Renommer la nouvelle table avec l'ancien nom;

Exemple : Supposons que nous ayant une table T_Person dont la définition est :

```
CREATE TABLE Tmp_T_PERSONNE
(
PER_ID int NOT NULL,
PER_NOM varchar(50) NOT NULL,
PER_PRENOM varchar(50) NULL,
PER_NE_LE smalldatetime NOT NULL,
)  ON [PRIMARY]
GO
--Et que nous voulons changer le type Per_Nom du type varchar(50) au type
varchar(100)
--Nous aurons :
BEGIN TRANSACTION
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
--Créer une table temporaire ayant même structure que la première
CREATE TABLE Tmp_T_PERSONNE
(
PER_ID int NOT NULL,
PER_NOM varchar(100) NOT NULL,
PER_PRENOM varchar(50) NULL,
```

```
PER_NE_LE smalldatetime NOT NULL,
)  ON [PRIMARY]
GO
-- Peupler la table
IF EXISTS(SELECT * FROM T_PERSONNE)
 EXEC('INSERT INTO Tmp_T_PERSONNE (PER_ID,PER_NOM, PER_PRENOM, PER_NE_LE,
PAY_ID, PER_NE_A)
SELECT PER_ID, PER_NOM, PER_PRENOM, PER_NE_LE FROM T_PERSONNE TABLOCKX')
GO
--Supprimer la table
DROP TABLE dbo.T_PERSONNE
GO
--Renommer la nouvelle table avec l'ancien nom
EXECUTE sp_rename N'Tmp_T_PERSONNE', N'T_PERSONNE', 'OBJECT'
GO
COMMIT
```

Comment renommer une base de données ?

Pour renommer une base de données, MS SQL Server fournit la procédure stockée **sp_renamedb**. Exemple :

```
EXEC sp_renamedb('Test','UneBaseTest')
```

Vous pouvez également créer une nouvelle base de données, importer les données par DTS de l'ancienne base de données vers la nouvelle, puis supprimer l'ancienne base de données.

Comment visualiser le code T-SQL d'une procédure stockée ?

```
SELECT text
FROM dbo.syscomments, dbo.sysobjects
WHERE syscomments.id = sysobjects.id
And sysobjects.xtype = 'P'
AND sysobjects.name='MaProcédure'
```

Mais bien plus simplement, et avec les bons retour-chariots :

```
sp_helptext 'MaProcédure'
```

Comment lister les contraintes de clés primaires et étrangères des tables d'une base de données ?

```
SELECT *
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
WHERE TABLE_NAME = 'matable'
```

Comment trouver la liste des tables dont dépend la vue ?

```
SELECT DISTINCT NECESSAIRE.NAME
FROM SYSOBJECTS AS NECESSAIRE
INNER JOIN SYSDEPENDS AS DEPENDENCES
ON NECESSAIRE.ID = DEPENDENCES.depid
INNER JOIN SYSOBJECTS AS DEPENDANTE
ON DEPENDENCES.id = DEPENDANTE.id
WHERE DEPENDANTE.name='NOMDELAVUE'
```

Comment comparer 2 tables ?

```
SELECT s1.name, s1.type, s2.name, s2.type
FROM syscolumns s1, syscolumns s2
WHERE s1.id = object_id('MaTable1')
and s2.id = object_id('MaTable2')
and s1.name=s2.name
and s1.type<>s2.type
```

Comment trouver une table à travers toutes les bases ?

Voici une procédure permettant de rechercher toutes les bases contenant une table de nom **@SCH.@TAB** :

```
DECLARE @SCH NVARCHAR(128), @TAB  NVARCHAR(128);
SELECT @SCH = '???' , @TAB = '???';
DECLARE @SQL NVARCHAR(max)
SET @SQL = '';
SELECT @SQL = @SQL + 'SELECT * FROM '
       + name + '.INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA = ''' +
COALESCE(@SCH, 'dbo') + '''  AND TABLE_NAME = ''' + @TAB +''';'
FROM   sys.databases;
 EXEC (@SQL);
```

Auditer le taux d'occupation de vos disques de manière automatique ?

Voici un ensemble de codes SQL utilisant des procédures système et l'agent SQL pour scruter le taux d'occupation des disques et remonter une alerte en cas de dépassement.

Création des objets dans la base de données MSDB

Création des tables de suivi de l'évolution de l'espace disque et de leur taux d'occupation. Notez l'utilisation du schéma **S_SYS** dans msdb :

```
USE msdb;
GO
CREATE SCHEMA S_SYS

CREATE TABLE T_A_DISK_DSK
( DSK_ID         INT       NOT NULL PRIMARY KEY,
  DSK_UNIT       CHAR(1)   NOT NULL UNIQUE CHECK (DSK_UNIT COLLATE
French_CI_AS BETWEEN 'C' AND 'Z'),
  DSK_ALERT_PC   FLOAT     NOT NULL DEFAULT 30.0 CHECK (DSK_ALERT_PC BETWEEN
0.0 AND 100.0))

CREATE TABLE T_A_TRACE_SPACE_DISK_TSP
( TSP_ID         INT       NOT NULL PRIMARY KEY,
  DSK_UNIT       CHAR(1)   NOT NULL FOREIGN KEY REFERENCES T_A_DISK_DSK
(DSK_UNIT),
  TSP_DATETIME   DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  TSP_SIZE_MO    INT       NOT NULL,
  TSP_USED_MO    INT       NOT NULL);
GO

CREATE INDEX X_TSP_DTM ON S_SYS.T_A_TRACE_SPACE_DISK_TSP (TSP_DATETIME,
DSK_UNIT);
GO
```

Création de la procédure de capture des données d'espace disque :

```
CREATE PROCEDURE S_SYS.P_AUDIT_SPACE_DISK
AS

SET NOCOUNT ON;

DECLARE @HDL int,
        @FSO int,
        @HD char(1),
        @DRV int,
        @SZ varchar(20),
        @MB bigint ;

SET @MB = 1048576;

CREATE TABLE #HD (HD_UNIT      char(1) PRIMARY KEY,
                  HD_FREESPACE int NULL,
                  HD_SIZE      int NULL);

INSERT INTO #HD (HD_UNIT, HD_FREESPACE)
EXEC master.dbo.xp_fixeddrives;

DELETE FROM #HD
WHERE  HD_UNIT NOT IN (SELECT DSK_UNIT
                       FROM   S_SYS.T_A_DISK_DSK);

EXEC @HDL = sp_OACreate 'Scripting.FileSystemObject',@FSO OUT;
IF @HDL <> 0 EXEC sp_OAGetErrorInfo @FSO;

DECLARE C CURSOR LOCAL FAST_FORWARD
FOR SELECT HD_UNIT
    FROM   #HD;

OPEN C;

FETCH NEXT FROM C INTO @HD;

WHILE @@FETCH_STATUS=0
BEGIN

        EXEC @HDL = sp_OAMethod @FSO, 'GetDrive', @DRV OUT, @HD
        IF @HDL <> 0 EXEC sp_OAGetErrorInfo @FSO;

        EXEC @HDL = sp_OAGetProperty @DRV, 'TotalSize', @SZ OUT
        IF @HDL <> 0 EXEC sp_OAGetErrorInfo @DRV;

        UPDATE #HD
        SET    HD_SIZE = CAST(@SZ AS FLOAT) / @MB
        WHERE  HD_UNIT = @HD;

        FETCH NEXT FROM C INTO @HD;

END

CLOSE C;
DEALLOCATE C;

EXEC @HDL=sp_OADestroy @FSO;
IF @HDL <> 0 EXEC sp_OAGetErrorInfo @FSO;
```

```
INSERT INTO S_SYS.T_A_TRACE_SPACE_DISK_TSP (TSP_UNIT, TSP_SIZE_MO,
TSP_USED_MO)
SELECT                                      HD_UNIT,  HD_SIZE,     HD_SIZE
- HD_FREESPACE
FROM   #HD

DROP TABLE #HD;

RETURN;
GO
```

<u>Mise en place dans l'agent SQL Server serveur d'une routine journalière de scrutation à 5h du matin</u>

```
CREATE PROCEDURE S_SYS.P_AUDIT_SPACE_DISK
AS

SET NOCOUNT ON;

DECLARE @HDL int,
        @FSO int,
        @HD char(1),
        @DRV int,
        @SZ varchar(20),
        @MB bigint ;

SET @MB = 1048576;

CREATE TABLE #HD (HD_UNIT      char(1) PRIMARY KEY,
                  HD_FREESPACE int NULL,
                  HD_SIZE      int NULL);

INSERT INTO #HD (HD_UNIT, HD_FREESPACE)
EXEC master.dbo.xp_fixeddrives;

DELETE FROM #HD
WHERE  HD_UNIT NOT IN (SELECT DSK_UNIT
                       FROM   S_SYS.T_A_DISK_DSK);

EXEC @HDL = sp_OACreate 'Scripting.FileSystemObject',@FSO OUT;
IF @HDL <> 0 EXEC sp_OAGetErrorInfo @FSO;

DECLARE C CURSOR LOCAL FAST_FORWARD
FOR SELECT HD_UNIT
    FROM   #HD;

OPEN C;

FETCH NEXT FROM C INTO @HD;

WHILE @@FETCH_STATUS=0
BEGIN

        EXEC @HDL = sp_OAMethod @FSO, 'GetDrive', @DRV OUT, @HD
        IF @HDL <> 0 EXEC sp_OAGetErrorInfo @FSO;

        EXEC @HDL = sp_OAGetProperty @DRV, 'TotalSize', @SZ OUT
        IF @HDL <> 0 EXEC sp_OAGetErrorInfo @DRV;
```

```
        UPDATE #HD
        SET    HD_SIZE = CAST(@SZ AS FLOAT) / @MB
        WHERE  HD_UNIT = @HD;

        FETCH NEXT FROM C INTO @HD;

END

CLOSE C;
DEALLOCATE C;

EXEC @HDL=sp_OADestroy @FSO;
IF @HDL <> 0 EXEC sp_OAGetErrorInfo @FSO;

INSERT INTO S_SYS.T_A_TRACE_SPACE_DISK_TSP (TSP_UNIT, TSP_SIZE_MO,
TSP_USED_MO)
SELECT                                       HD_UNIT,  HD_SIZE,     HD_SIZE
- HD_FREESPACE
FROM   #HD

DROP TABLE #HD;

RETURN;
GO
```

Vous devez remplacer "ServerSQL[\instance]" par le nom de votre serveur et SA par le compte de connexion sous lequel cette routine doit tourner.

Comment importer ou exporter un diagramme ?

Les informations sur les diagrammes sont stockées dans la table dtproperties dans chaque base de données. Voici une procédure pour transférer les diagrammes d'une base de données vers une autre :

1 - Faire un clic droit sur la base qui contient le schéma a transféré, cliquez sur « **Toutes les tâches** » puis sur « **Exporter des données** »

2 - Configurez la source puis cliquez sur le bouton « **Suivant** ».

3 - Configurez la base de destination et cliquez sur le bouton « **Suivant** ».

4 - Dans l'écran « **Spécifier Copie ou Interrogation de Table** », cliquez sur « **Utilisez une requête pour spécifier les données à transférer** », puis cliquez sur « **Suivant** ».

5 - Dans l'écran « **Saisie de l'instruction SQL** », tapez la requête suivante :

```
Select * From dtproperties
```

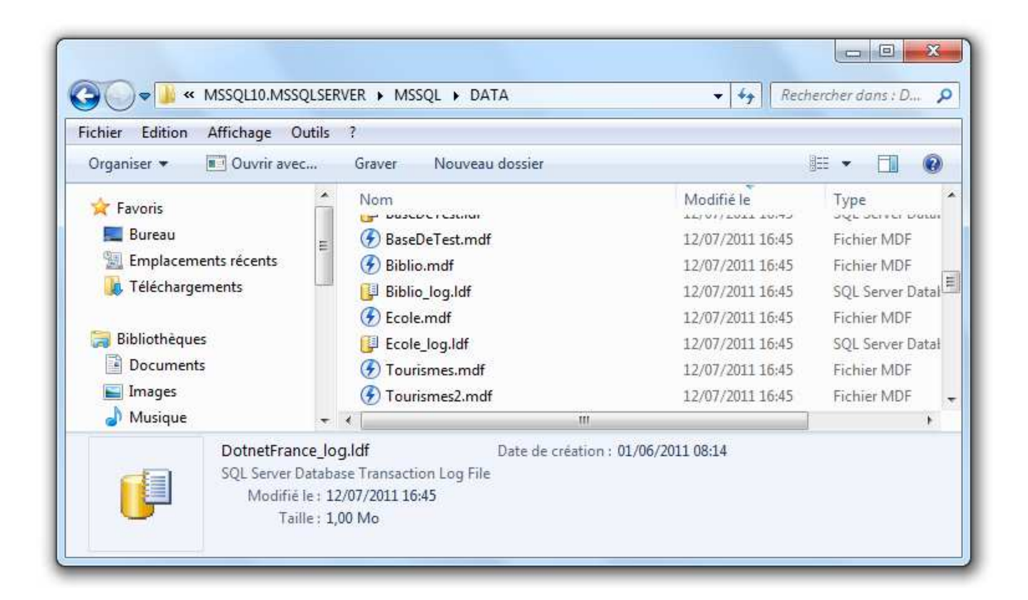
Puis cliquez sur Suivant.

6 - Dans l'écran « Sélectionner **les tables et les vues sources** », choisissez la table **dtproperties** dans la colonne destination puis cliquez sur « **Suivant** ».

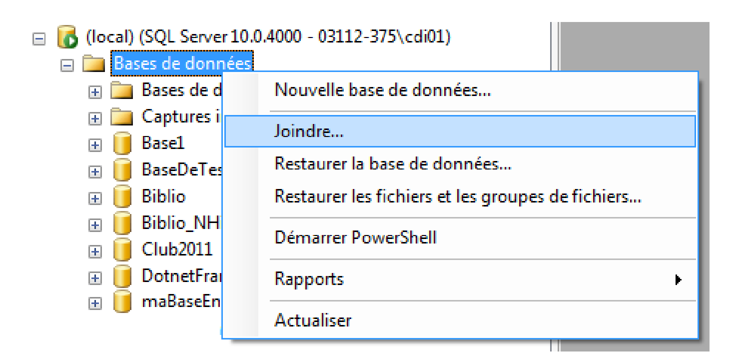
7 - L'écran « **Enregistrer, planifier et dupliquer le lot** », choisissez « **Exécuter immédiatement** », puis cliquez sur « **Suivant** » et enfin cliquez sur « **Terminer** ».

Comment joindre (importer) une base extérieure

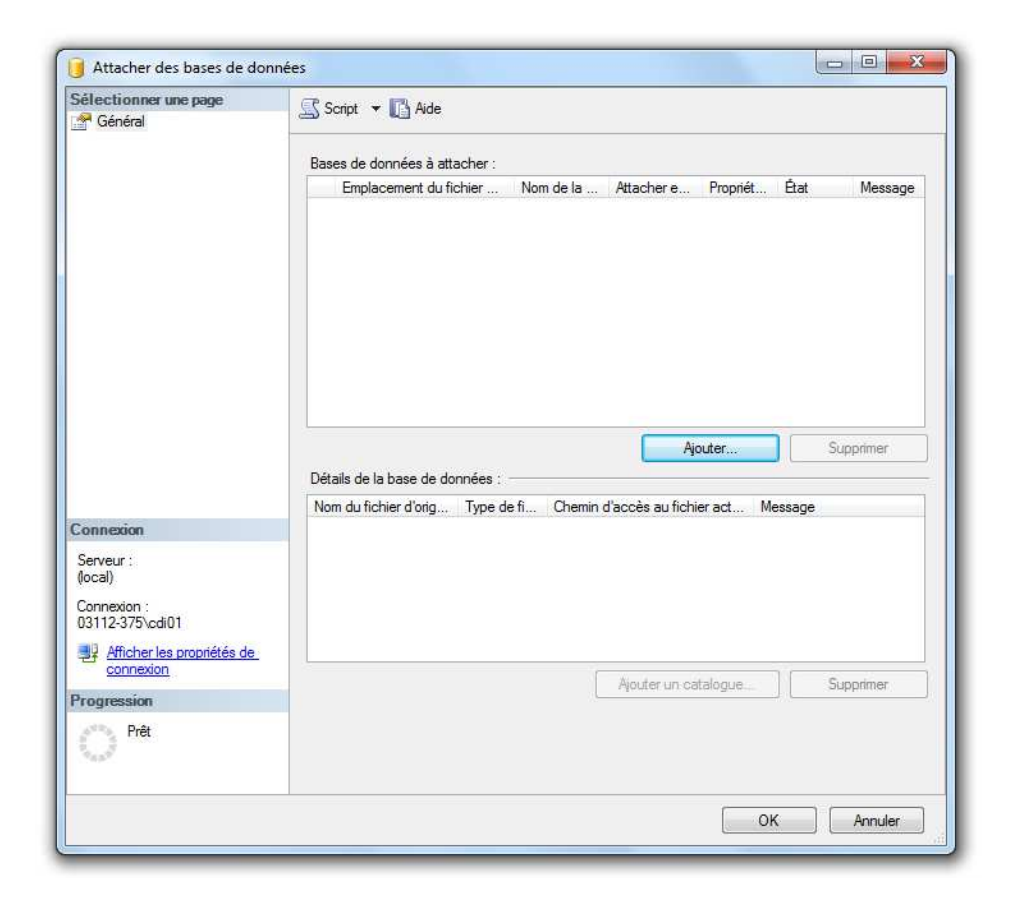
Une base de données se compose d'un fichier « **mdf** » et « **ldf** ». Si vous souhaitez importer une base de données déjà existante dans SQL Server, placer ces deux fichiers dans le dossier correspondant à la version de votre SQL Server. Dans notre cas, il s'agira du dossier « **C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA** ».



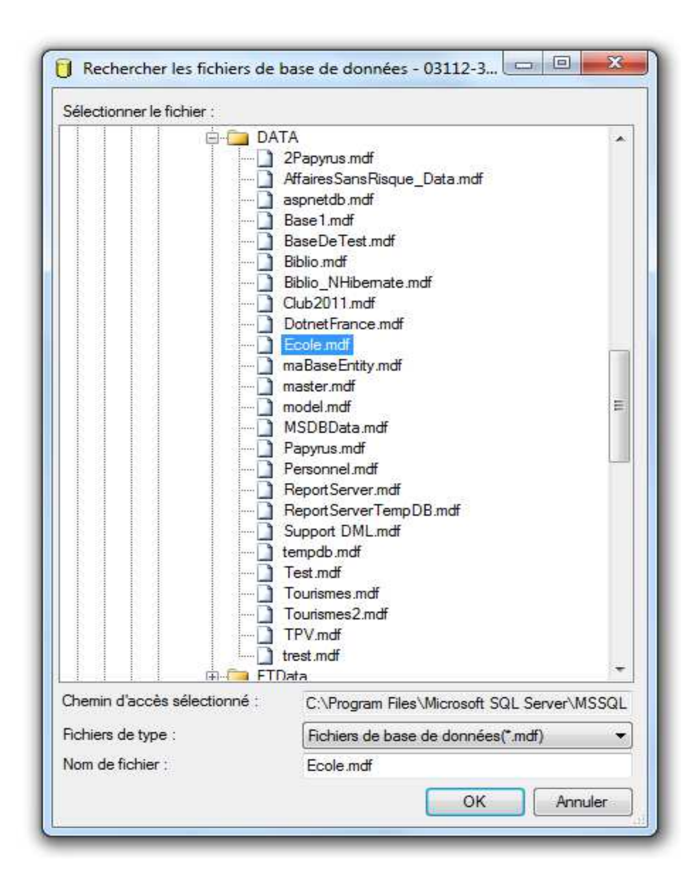
Dans « **SQL Server** », Sur le dossier «**Bases de données** », cliquez droit puis « **Joindre** ».



Dans la fenêtre qui apparaît à votre écran, cliquez sur « **Ajouter** ».



Recherchez les fichiers correspondant à votre base puis cliquez sur « **Ok** ».



Cliquez de nouveau sur « **Ok** ».

Attacher des bases de données

Sélectionner une page
Général

Script | Aide

Bases de données à attacher :

Emplacement du fichie...	Nom de la ...	Attacher e...	Propriét...	État	Message
C:\Program Files\...	9CEF792...	9CEF792...	03112-...		

Ajouter... | Supprimer

Détails de la base de données « 9CEF792E751E8274FF6062DAF6E3D9F... » :

Nom du fichier d'ori...	Type de fic...	Chemin d'accès au fichier ac...	Message
Ecole	Données	C:\Program Files\Micro...	
Ecole	Journal	C:\Program Files\Micro...	

Ajouter un catalogue... | Supprimer

Pour mettre à niveau un catalogue de texte intégral, cliquez sur Ajouter un catalogue, puis recherchez-le et sélectionnez-le. Les index de recherche en texte intégral seront importés, en fonction de l'option de mise à niveau du catalogue de texte intégral.

Connexion

Serveur :
(local)

Connexion :
03112-375\cdi01

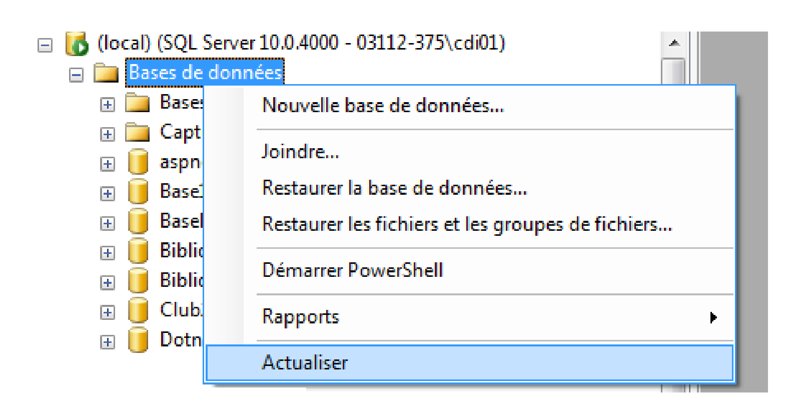
Afficher les propriétés de connexion

Progression

Prêt

OK | Annuler

Pour terminer, cliquez sur « **Actualiser** ».





AFPA 2011

Conception et Réalisation
D'une base de données
Merise • PowerAMC • SQL Server • T-SQL



Réaliser une base de données sous SQL Server

La méthode Merise est une méthode d'analyse, de conception et de réalisation de systèmes d'informations informatisés. Power AMC est un logiciel de modélisation. Microsoft SQL Server est un système de gestion de base de données.

Ce tutoriel se présente sous forme d'ouvrage avec pour objectif la réalisation d'une base de données sous Microsoft SQL Server en passant par la conception à l'aide de la méthode d'analyse Merise sous Power AMC. L'ouvrage se destine exclusivement aux étudiants de la formation professionnelle de l'Afpa, qui souhaitent apprendre et comprendre les grandes étapes nécessaires à la conception et à la réalisation d'une base de données.

Stéphane Grare

Concepteur et développeur C#, Stéphane Grare, passionné par la programmation informatique, est l'auteur de plusieurs tutoriels et livres blancs sur différents langages de programmation informatiques. Il est aussi à l'initiative du projet Simply, une gamme d'applications dédiées à la bureautique.

Au Sommaire

Merise • Introduction à la méthode • Conception avec Power AMC • **Créer la base de données** • Création de la base • Création de tables • Modifications de tables et contraintes • Supprimer une table • Supprimer une base • Groupes de fichiers • Les partitions • Schémas de la base • **Alimenter la base de données** • Saisir des données dans vos tables • Les index • Les vues • Les vues indexées • Gestion des schémas • Générer des scripts • **Sauvegarder et restaurer la base** • Sauvegarde • Restauration • **Sécurité de la base** • Gestion des accès serveur • Gestion des connexions • Modification des connexions • Suppression des connexions • Gestion des utilisateurs • Modification des utilisateurs • Suppression des utilisateurs • Gestion des droits • Droits • Les rôles • **Programmations SGBD** • Le Langage DML • Les fonctions utilisateur • Procédures Stockées • Les curseurs • Les transactions et les verrous • Les déclencheurs • Déboguer

À qui s'adresse cet ouvrage ?

- Aux étudiants de la formation Concepteur Développeur
- Aux étudiants de la formation Développeur Logiciels



Sur le site www.afpa.fr

- La formation professionnelle
- Valider ses acquis
- Se remettre à niveau

